August 5, 1994

To the Graduate School:

This thesis entitled "A Reconfigurable Multiprocessor Architecture and its Arithmetic Performance" and written by Mr. Kenneth B. Winiecki, Jr. is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

_____

Thesis Advisor

We have reviewed this thesis
and recommend its acceptance:

_____

_____

Accepted for the Graduate School:

_____

A RECONFIGURABLE MULTIPROCESSOR ARCHITECTURE AND ITS ARITHMETIC PERFORMANCE

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

---

by

Kenneth B. Winiecki, Jr.

August 1994

ABSTRACT

A word-wide processing element ("PE") based on an existing 1-bit-wide design is developed for a high-performance, massively-parallel, rectangular-mesh-connected, globally-routed, reconfigurable, MSIMD computer architecture. The PE can perform up to three operations per instruction cycle (one transfer and two Boolean), and an instruction can be executed in one clock cycle. It can communicate with up to four other PEs and a global data router via three word-wide ports and two bit-wide ports. Individual PE operation can be disabled by the controller and conditionally by the PE itself. Carry-lookahead logic facilitates fast full-addition/subtraction, and variable-shift registers enable improved floating-point performance. Configurations of PEs are developed to perform integer and floating-point arithmetic with various methods and degrees of parallelism. Behavioral models of the PE and the configurations are developed in the Verilog hardware description language, and the performance of the configurations is observed. It is found that the PE design facilitates significant parallelization of many arithmetic operations, thereby producing appreciable speedup. The success of this research indicates that this PE design should be further considered for the basis of a high-performance computer architecture.

DEDICATION

This work is dedicated to my parents, Mr. and Mrs. Kenneth B. and Mary Winiecki.

ACKNOWLEDGMENTS

I wish to acknowledge the support, friendship, and patience of my advisor, Dr. W. B. Ligon, III, and of my colleagues, Mr. K. O. Wichmann, Ms. L. L. Joiner, and Mr. D. C. Stanzione, Jr.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Background and Motivation

The success of humanity as a species is attributable in large part to its striving to exert control over its environment. Since control requires feedback, and feedback is information, a critical factor in the development of control is the development of the ability to process information. Down through the ages, information processing tools were continuously augmented: from the human brain alone, to counting-stones, to written language, to the abacus, to the mechanical tabulating machine, to the electrical adding machine, and finally, to the binary electronic computer. However, the amount of environmental control achieved by any of these improvements is only a minuscule portion of what can be conceived and thus what is desired. This corresponds to the fact that the amount of information that exists to be processed is consistently far greater than mankind is able to process, a fact which relentlessly drives the further improvement of information processing.

One of the current and most apparent manifestations of this drive is the tremendous development in the area of computers. The technology of computation equipment evolved from relays, to vacuum tubes, to discrete transistors, to integrated circuits. Similarly, the methodology of computing evolved from sequential processing to simultaneous or "parallel" processing, this being the only means available of increasing the processing of information, i.e., improving processing performance, given a particular set of technological limits

at a particular time. Sequential processing is not the focus of this research, and so is not further discussed here.

At one extreme, the simplest form of parallel processing is "multicomputing", the simultaneous operation of multiple computers. This obviously increases processing performance by solving multiple independent problems simultaneously, and recently through the advent of distributed operating systems, also addresses the solution of larger or longer singular problems. At the other extreme is the form of parallelism known as "pipelining", the parallel execution of the multiple stages of a single processor instruction. This has the effect of reducing a problem's execution time, thereby also addressing both types of processing performance improvement. Today, pipelining is used in virtually all high-performance processor designs. In between the two extremes is the form of parallelism exhibiting the most diversity, "multiprocessing". Multiprocessing has two meanings: the application of multiple simultaneously-operating processors to a single problem (regardless of the computer in which they reside), or the application of multiple simultaneously-operating processors within a single computer (regardless of the number of problems to which they are applied). The former is a form of multicomputing, which is not the focus of this research and so is not further discussed here; use of the term multiprocessing hereafter refers to the latter.

Depending on the way multiple processors are applied, they can be used to solve multiple small, independent problems and/or a large or long singular problem. Forms of multiprocessing can be classified by two characteristics: the number of simultaneous streams of instructions, single or multiple, and the number of simultaneous streams

of data, single or multiple. This provides a total of four classifications: single-instruction-stream / single-data-stream ("SISD"), single-instruction-stream / multiple-data-stream ("SIMD"), multiple-instruction-stream / single-data-stream ("MISD"), and multiple-instruction-stream / multiple-data-stream ("MIMD"). This classification was introduced and discussed by Flynn [1].

SISD is useful for robustness, but the multiple processors provide no increase in the processing performance of a problem over a single processor since they all execute exactly the same operations on exactly the same values. Of course, the remaining three classifications are supersets of SISD, so they all have the potential to provide robustness. MISD increases processing performance for problems that require multiple operations to be executed on the same data elements, a somewhat rare type of problem, judging by the lack of discussion found in the literature. SIMD increases processing performance for problems that require the same operation to be performed on different data elements, a very common type of problem that includes scientific applications such as fluid dynamics, engineering applications such as computer-aided circuit design and development, artificial intelligence applications such as pattern recognition, and general applications such as database searching. The first major SIMD computer was the ILLIAC-IV [2], and others that followed include the ICL DAP [3], Goodyear MPP [4], TMC CM-1 [5], and IBM GF11 [6]. Most of the architectures mentioned in this paper are also discussed in [7]. The MIMD paradigm is the most versatile; it is a superset of SIMD and MISD, and so has the potential to increase the performance of the same types of problems. Because each processor receives a separate stream of instructions, MIMD can also

support the processing of multiple small independent problems. There are numerous subclassifications and examples of the MIMD paradigm, but these are not the focus of this research and are not discussed here.

Unfortunately, the classification of paradigm is commonly obscured by "hybridizing", where a system is designed with characteristics of more than one paradigm. For one example, a processor could be designed with an instruction which could, depending on some condition internal to the processor, cause it to be disabled (or enabled). Since in the SIMD paradigm different processors receive different data streams and can therefore generate different internal conditions, a SIMD computer constructed with this processor would have a MIMD characteristic in that the conditions of some processors would allow them to be disabled while the conditions of others would not, so all the processors would not be executing the all the same instructions. In fact, this is the case with all of the SIMD and MSIMD architectures mentioned here, the reason being that real applications without data-dependent iterative processes are exceedingly rare. Because of the universality of the MIMD-like disable feature in otherwise-SIMD system designs, such systems are still referred to as SIMD.

A second form of paradigm hybridizing is the distribution of processor control among individual processors or subgroups of processors. For example, a computer design could be based on the SIMD paradigm but modified so that instead of the system controller directly controlling all the processors, it controls a set of sub-controllers, each of which controls a different sub-group of the processors. Alternately, a computer design could be based on the MIMD paradigm but modified so that instead of each processor controlling its own

execution, control is concentrated in a two-layer controller hierarchy consisting of a single master controller and a number of slave controllers which govern the operation of groups of processors. These two designs describe the same system, one which exhibits SIMD behavior within a processor group but MIMD behavior between groups. The resulting paradigm could thus be termed multiple-single-instruction-stream / multiple-data-stream ("MSIMD"), which is clearly another subset of MIMD.

The flexibility provided by MIMD and MSIMD makes possible a unique processing approach called "reconfigurability" or "multigauge processing", where the number of processors working on one problem can be changed for another problem, and the level of problem that can have different configurations can be as small as a machine operation. For example, if an application reaches a point where 2000 integers are to be multiplied with 2000 other integers to produce 2000 results on a machine with 4000 processors, and each processor can perform a multiplication operation in 300 cycles, it is clear that the 2000 operations would be performed in 300 cycles. However, if this machine somehow supports a situation where every pair of processors is made to collaborate on a single multiplication operation and perform it in 200 cycles, the 2000 operations could be performed in 200 cycles. On the other hand, if the application instead calls for 3000 multiplication operations, then the dual-PE "configuration" would require 400 cycles because it can only do 2000 at a time, while the singular-PE configuration would still require only 300 cycles. Thus the ability to support different configurations of PEs adds an opportunity to increase performance via a tradeoff

between the time required to perform an operation and the number of simultaneous operations that can be performed.

Reconfigurability can be accomplished either through software, i.e., the programming of each processor, or through hardware, by switching processor interconnections. It can also be either dynamic, i.e., able to change in the middle of a some predefined unit of processing, or static, i.e., fixed for that unit of processing, although this distinction is trivial for software reconfigurability. The concept of reconfigurability was first developed by Kartashev [8][9], discussed as multigauge processing by Snyder [10] and as reconfigurable mesh computation by Miller [11], and evaluated by Ligon [12][13]. A number of hardware-reconfigurable architectures have been developed, including the CHiP [14], PASM [15], TRAC [16], NETRA [17], and PPA [18]. Hardware reconfigurability is not a focus of this research, however, and so is not further discussed here. There do not exist as many examples of software reconfigurability, the two major ones being the Goodyear STARAN [19] and TMC CM-2 [20]. In software reconfigurability, for example, each processor could be programmed to execute the same instructions for a particular set of instruction cycles, producing SIMD operation. Then for the next set of instruction cycles, they could be programmed to each execute different instructions, effecting a reconfiguration to produce MIMD operation. Then for the next set of instruction cycles, they could be programmed as if all the processors were divided into groups, and each processor within a group executes a different instruction while each set of corresponding processors from among the groups executes the same instruction. This effects another reconfiguration, this time

producing SIMD behavior within a processor group but MIMD behavior between groups, i.e., a MSIMD system.

Furthermore, depending on the processor design, reconfigurability can be used as a means to two different ends: to affect the "precision" used to compute an operation, and to affect the "method" used to compute an operation. Precision reconfigurability is the ability to change the maximum size of the variables of an arithmetic computation. For example, say the processor was designed to be 1 bit wide, but a Boolean inversion of a 32-bit value is desired. A single processor might have to perform 32 sequential inversion operations to produce the result, but if the processors were precision-reconfigured as described above to operate in groups of 32 and the data was distributed appropriately among the streams, the time required would only be that of a single bit inversion operation. Note that this may require additional hardware facilities to support shifting and carry-calculation as well as the determination of conditions such as zero and overflow. Precision reconfigurability is clearly the goal of 1-bit-wide processor systems, and is an option for word-wide processor systems. Many of the systems mentioned previously use precision reconfigurability with 1-bit-wide processors: the ICL DAP, Goodyear MPP and STARAN, and TMC CM-1 and CM-2, for example. Many others use it with word-wide processors: the ILLIAC-IV, IBM GF11, TRAC, CHiP, and PASM.

Method reconfiguring is also called "capability" reconfiguring, but this is somewhat ambiguous because reconfiguring both the precision and the method affects the capability of the machine to perform a particular amount of work in a particular amount of time. Method reconfigurability, for example, might be used for floating-point value

normalization, which involves repeatedly shifting the mantissa left and incrementing the exponent (for more details, refer to the section, "Arithmetic Performance"). A single processor might have to perform these two processes sequentially, but if the processors were method-reconfigured as described above, one processor could perform the shift while another simultaneously performed the addition. This may require a small amount of overhead to first distribute the operands and then collect the results into one PE, but more than likely it will reduce the time required to perform the normalization operation.

Another way to classify multiprocessing systems is by "granularity", the relative size of the smallest activity that can be parallelized. Listed in order from "coarsest" to "finest", the activities are: programs, subroutines, statements, expressions, operations, instructions, and microinstructions. For example, multicomputing is generally the parallelization of programs, and pipelining is the parallelization of instruction stages or microinstructions. In the SIMD and MSIMD paradigms, parallelization generally occurs on the instruction level, and so finer-grained activity is not directly controlled by the system. Unfortunately, the classification of granularity is commonly obscured by the differing amounts of activity that can be defined for an instruction. On one extreme, for example, a processor could be designed to perform a high-level language statement as an instruction, so a computer built of such processors could equally be called statement-level SIMD and instruction-level SIMD. Similarly on the other extreme, a processor could be designed so that an instruction only performs the activity of a microinstruction, so a computer built of such processors could equally

be called instruction-level SIMD or microinstruction-level SIMD. In this case, solitary use of the term instruction-level SIMD is meaningless.

Objective and Tasks

This research follows along the lines of the ICL Distributed Array Processor ("DAP") [3], the Goodyear Massively Parallel Processor ("MPP") [4], and the Thinking Machines Corporation Connection Machine ("CM-1") [5] mentioned earlier. These are all massively-parallel, rectangular-mesh-connected, precision-reconfigurable SIMD machines intended for image processing (DAP and MPP), artificial intelligence (CM-1), and other applications characterized by large arrays of homogeneous data.

SIMD operation is desirable for two reasons. First, the price of MIMD's higher level of flexibility over SIMD is a higher level of complexity, both of hardware and software, thus incurring higher cost and a longer design cycle. Second, the processing of large arrays of homogeneous data cannot take advantage of much of the additional flexibility that MIMD can provide over SIMD, resulting in waste.

Precision reconfigurability is desirable because it facilitates maximum resource utilization when processing sets of data that have different precisions, thereby minimizing cost and maximizing flexibility and performance. This is an important ability since the precision of data elements such as pixel color is likely to differ from application to application. For example, reconfiguring a system to reduce precision could avoid the waste of processing power that would occur if the data word of the processor ("processing element", or "PE") was wider than the data element itself. A precision reduction could also produce a performance gain if performing an operation with a larger number of

lower-precision steps required fewer clock cycles than performing the same operation with a smaller number of larger-precision steps. Clearly, ultimate precision reconfigurability is achieved using 1-bit PEs, which all three of the machines do. 1-bit PEs have the additional advantages of being simple, small, inexpensive, and fast.

A rectangular mesh (North-East-West-South or "NEWS") interconnection topology is desirable because virtually all images are composed of rectangularly-positioned elements. The DAP and MPP meshes can be edge-connected via software to form a circular or spiral cylinder or torus, while the CM-1 adds a completely separate daisy-chain interconnection. The spiral and daisy-chain topologies support the view of processing data as a bit-stream. Additionally, the CM-1 provides a third completely separate interconnection topology in the form of a multidimensional hypercube routing network with connections to each PE. This accords it a level of flexibility far above that of the DAP or MPP.

In previous work by Ligon [21], a 1-bit PE was developed in an attempt to improve upon the one used in the CM-1 [5]. Both PEs are designed to perform two arbitrary Boolean functions of three variables, but Hillis' uses two memory operands and one memory result and requires 3 clock cycles per instruction cycle, while Ligon's operates register-to-register and requires only 1 clock cycle per instruction cycle, and adds a separate interface so that a memory-register transfer operation can be performed in parallel with the Boolean operations. In other respects, i.e., connectivity and disabling, the PEs are similar since the intended system architecture is the same. One relatively minor exception is the omission of the daisy-chain connections from Ligon's

design, as it was intended that the mesh would be flexibly edge-connectable in the manner of the MPP.

Since then, one major issue has become apparent. It was found that a large number of popular SIMD applications required floating-point arithmetic. This was not anticipated by TMC, and it served to highlight the CM-1's inadequate floating-point performance. Although the MPP predated the CM-1, it performed floating-point arithmetic much faster because each "1-bit" PE included a 30-bit variable-shift register. TMC attempted to rectify this in the CM-2 by restructuring the system to accommodate an optional floating-point package consisting of 2048 sets of off-the-shelf 64-bit floating-point coprocessors and custom 32 x 32-bit transposers, one set for every group of 32 PEs [20]. The success of this solution, however, was diminished somewhat by its increased cost and complexity. It also encouraged the programmer to view the CM-2 as a collection of 2048 word-wide PEs instead of 64K 1-bit PEs. These two factors, along with vast improvements in microprocessor price per performance ratio, subsequently compelled TMC to abandon the custom 1-bit PE and floating-point support hardware, adopt markedly less expensive off-the-shelf 64-bit microprocessors to accompany the floating-point coprocessors, increase the number of processors to 16K, and switch to the MIMD paradigm. Thus was produced the CM-5 [22].

Subsequent work by Ligon evaluating reconfigurability [12][13] revealed a possible alternate evolution for the massively-parallel, rectangular-mesh-connected, globally-routed, precision-reconfigurable SIMD architecture.

1. Give the programmer the more natural word-wide view by using word-wide PEs (sacrifice precision-reconfigurability to do so).

2. Retain the simplicity, small size, low cost, and speed of Ligon's 1-bit PE by designing a word-wide PE based on the same architecture.

3. Facilitate fast integer and floating-point arithmetic by providing the PE with carry-lookahead logic and variable-shift capability.

4. Increase the general opportunity for parallelism and facilitate additional speedup of arithmetic operations by providing method-reconfigurability.

5. Retain as much of the simplicity and low cost of the rectangular-mesh-connected, globally-routed, SIMD system architecture as possible by adopting the MSIMD paradigm.

The objective of the research presented in this paper is to evaluate the efficacy of these architectural characteristics for producing efficient sequential arithmetic operations and facilitating the parallelization and speedup of such operations. The tasks are to first design and model the PE, then design and model configurations of PEs for performing some common arithmetic operations, and finally obtain and analyze the performance results. These tasks are accomplished by modeling the behavior of the PE and configurations in the Verilog hardware description language ("Verilog HDL").

The remainder of the text is organized into four sections. The following section entitled "The Processing Element" describes the design and modeling of the PE. The section entitled "Arithmetic Performance" details the design, modeling, and performance of configurations of 1, 2, 4, and 8 PEs applied to integer multiplication and division and floating-point addition/subtraction and multiplication for both 32- and 64-bit data word widths. It also discusses the evolution of the PE design and how it was driven by the development of the configurations. The Analysis presents the performance results in various tabular and graphical forms and provides additional discussion. Finally, the Conclusion summarizes the research and presents possible directions of

future work. Following the text are the Appendix, containing complete Verilog program listings and sample output, and the List of References.

THE PROCESSING ELEMENT

Overview

The PE consists functionally of four major elements: a memory, a register set, and two logic units. The elements all have bit-widths equal to that of the data word ("W"), and they are controlled by signals which make up the microinstruction. Data communication is done through special registers. The arrangement is illustrated in Figure 1.

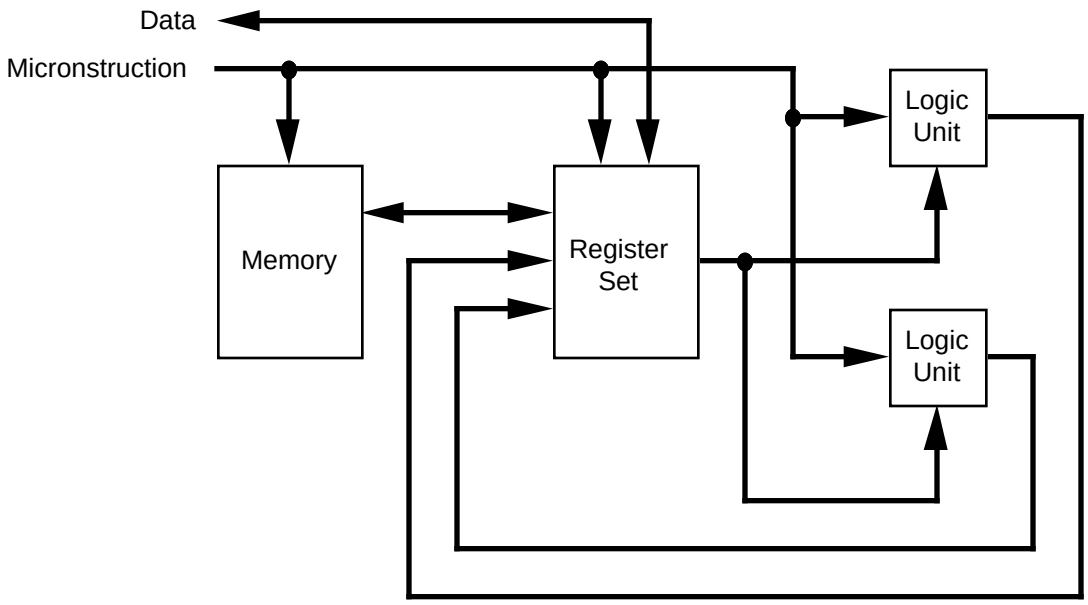


Figure 1. Simplified Processing Element Architecture

The PE can perform up to three tasks in a microinstruction cycle: one memory-register transfer operation, and two register-register logic/communication operations. The simplicity of this design allows the microinstruction to be executed in one clock cycle, providing one of the major advantages of this architecture.

The PE microinstruction reflects the simplified architectural view. It is divided into three major fields governing the memory, logic/communication, and disable functions, as depicted in Figure 2.

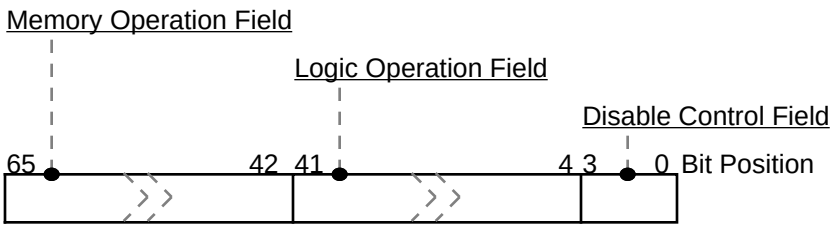


Figure 2. Generalized Microinstruction Format

As shown in the figure, the microinstruction is 66 bits wide: 46 for control signals and 20 for PE memory addressing. The memory address width is arbitrary; 20 bits is shown here merely for purposes of illustration. For the same reason, W (the data word width) is 32 bits, making the PE memory size 4 MB, or 1 MW.

The three subsections immediately following describe the PE architecture using the organization just presented: first memory, then logic, and finally disable. This section then concludes with a subsection detailing the behavioral model of the PE. Usage of the PE is presented in the next section, "Arithmetic Performance".

Memory-Register Architecture

Memory operation is straightforward. All memory interactions take place through data paths of width W, and with only three registers of the register set, the main registers IN1, IN2, and OUT; this design was intended to keep the implementation as simple as possible. Data transfer is controlled by four bits in the Memory Operation Field, with the remainder of the field comprising the memory address.. This arrangement is shown in Figure 3.

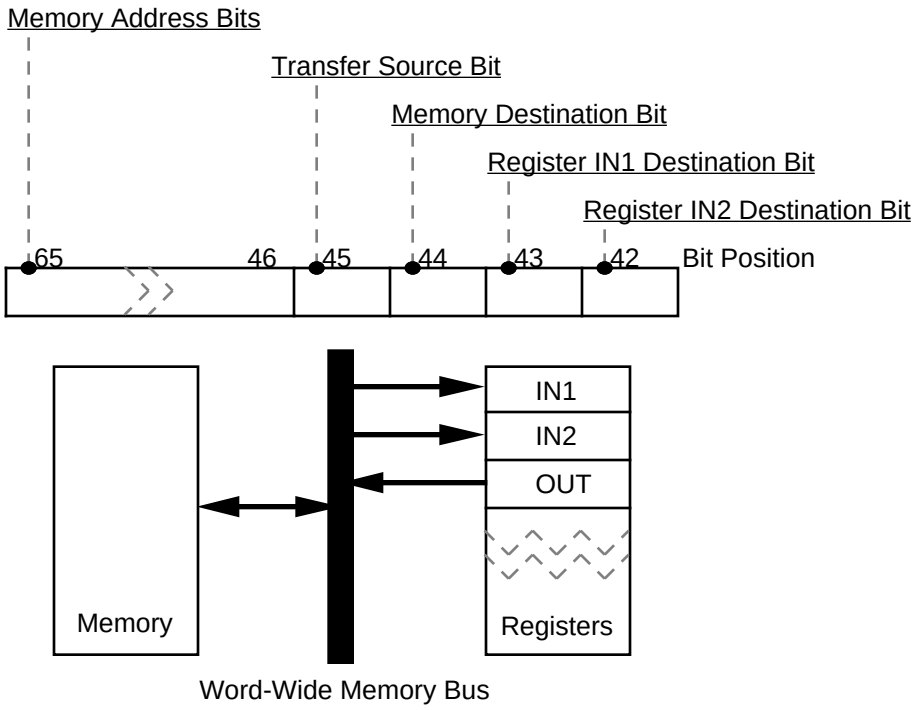


Figure 3. Memory-Register Architecture

The data transfer control bits consist of one source and three destination specification bits. The Transfer Source Bit selects between the two possible data sources that can be latched onto the bus: the OUT register is indicated by a 0, and memory is indicated by a 1. The latching of the bus input is timed to occur before any possible modification of the OUT register by other PE activity during the microinstruction cycle. The three destination bits determine which units will subsequently latch the data off of the bus: memory is indicated by setting the Memory Destination Bit, the IN1 register is indicated by setting the Register IN1 Destination Bit, and the IN2 register is indicated by setting the Register IN2 Destination Bit. The latching of the bus output is timed to occur after any possible use of the IN1 and IN2 registers by other PE activity during the microinstruction cycle.

One feature of this design is the ability to specify multiple destinations for the data, thereby allowing a certain form of parallel data transfer. The related ability to specify no destinations provides the mechanism for a null memory operation (memory "NOP"). The microinstruction architecture also appears to permit memory to be specified as both a source and a destination, but because memory is not expected to be fast enough, such a specification is ignored. A comprehensive list of memory-register data transfer operations consists of 11 entries, and is presented in Table I.

The architecture described supports direct addressing. To be able to write widely-applicable microprograms, however, some form of indirect addressing is required. No such ability is included in this PE design because it is not needed for the exploration of arithmetic performance, but the intention is that it should be added eventually.

Table I

Memory-Register Data Transfer Operations

Memory Address	Transfer Source	Memory Destination	Register IN1 Destination	Register IN2 Destination	Description
x	x	0	0	0	NOP
x	0	0	0	1	OUT -> IN2
x	0	0	1	0	OUT -> IN1
x	0	0	1	1	OUT -> IN1, IN2
<addr>	0	1	0	0	OUT -> mem[addr]
<addr>	0	1	0	1	OUT -> IN2, mem[addr]
<addr>	0	1	1	0	OUT -> IN1, mem[addr]
<addr>	0	1	1	1	OUT -> IN1, IN2, mem[addr]
<addr>	1	x	0	1	mem[addr] -> IN2
<addr>	1	x	1	0	mem[addr] -> IN1
<addr>	1	x	1	1	mem[addr] -> IN1, IN2

Logic/Communication Architecture

The register set consists of 20 word-wide elements: three main registers (IN1, IN2, and OUT), five communication registers (ROUTER, NORTH, SOUTH, EAST, and WEST), ten general-purpose registers (R0 - R9), a DISABLE register, and a CONDITION register. Two Boolean logic units and three special-purpose logic units complete the logic/communication architecture. The configuration is shown in Figure 4, sans control signals.

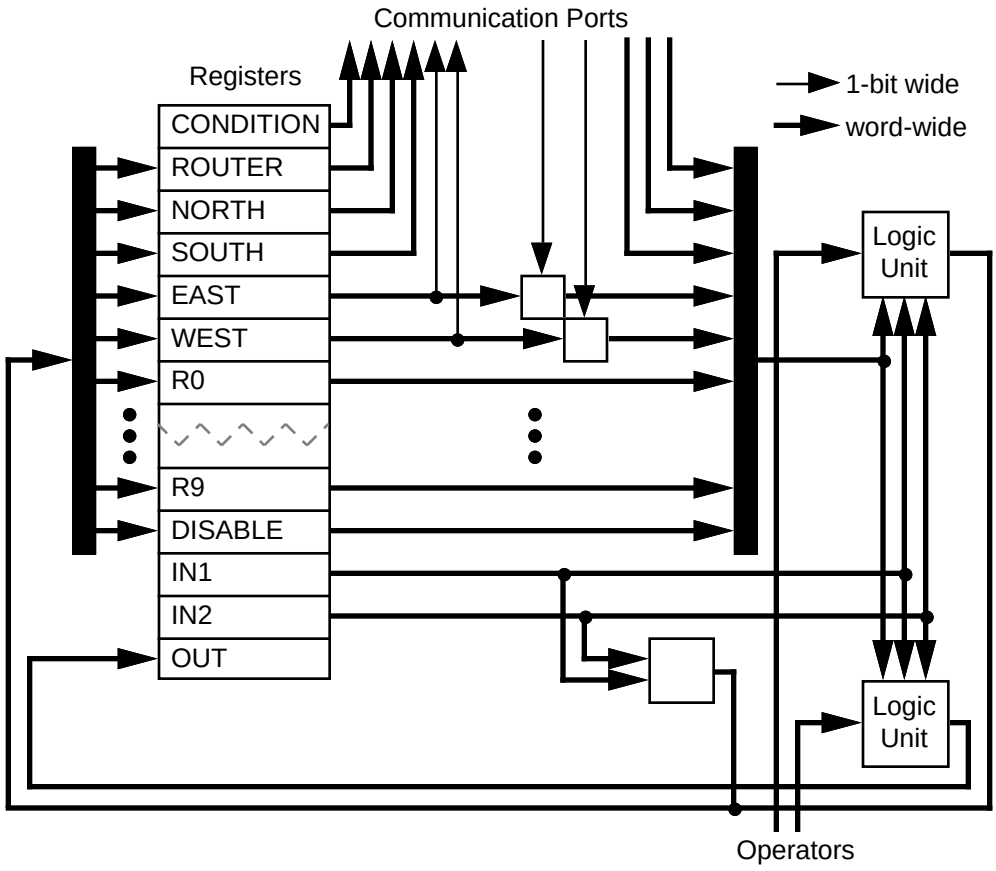


Figure 4. Logic/Communication Architecture

General operation proceeds as follows. The two Boolean logic units are each presented with four sources of data: the two main input registers (IN1 and IN2), the selectable source register, and the respective operator (from the microinstruction). One logic unit outputs its result to a selectable destination register; the other, to the main output register (OUT). Finally, condition flags are set in the CONDITION register. Communication takes place with four neighboring PEs (connected in the common rectangular-mesh fashion) and a global data router through designated "communication" registers; in general, loading from one actually loads from its corresponding input port, while storing to one makes the data accessible outside the PE through its corresponding output port. Certain useful functions that are beyond the capabilities of the Boolean logic units, namely shifting, carry-word computation, and disabling, are performed by special-purpose logic units.

The Logic Operation Field shown generally in Figure 2 consists of six subfields: Carry Control, Source Register, Shift Control, Register Operator, Destination Register, and OUT Operator. This format is depicted in Figure 5.

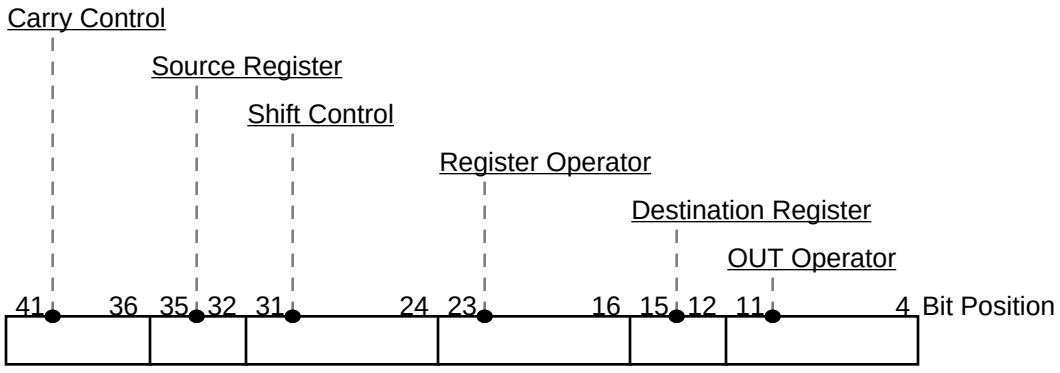


Figure 5. Logic Operation Field Format

The logic/communication function is best explained in a different order than is presented by the Logic Operation Field. Therefore, the register set is discussed first, along with the Source and Destination Register subfields. This is followed by a discussion of the shifting function and the Shift Control subfield. The logic function and the Register and OUT Operator subfields are discussed next. Lastly, a discussion of carry-word computation along with the Carry Control subfield is presented.

Register set and specification. The Source and Destination Register subfields shown in Figure 5 each consist of four bits containing the encoded identity of the desired selectable source or destination register. As indicated by the positions of the bus symbols in Figure 4, 16 of the registers are designated "selectable": the five communication registers, the ten general-purpose registers, and the DISABLE register. Note that the Destination Register subfield is only large enough to specify a single register, although the architecture could easily support multiple destinations at the expense of a wider microinstruction. The particular assignment of bit codes to registers can be seen in the discussion of the behavioral model as well as in the parameter section of any module program, all of which appear in the Appendix.

The choice of the number of general-purpose registers was not completely arbitrary. To be able to select the five communication registers and the DISABLE register, at least three bits are required. This provides eight codes, leaving two unused which can be assigned to general-purpose registers. However, two registers are not enough to allow efficient microprogramming; adding one more bit, for a total of

four, adds eight more codes, allowing for up to ten general-purpose registers. This has been found to be adequate for the programs required for this research. Should requirements change in the future, the simplicity of this design enables easy modification.

The communication registers are named for the "direction" that the data comes from when it is read. Note this is opposite of the direction that the data is made available to when it is written. For example, a PE's EAST register input port would be connected to its eastern neighbor's EAST output port, but the same PE's EAST output port would be connected to its western neighbor's EAST input port. This can initially be confusing, so Figure 6 is presented for clarification. It depicts five PEs, with the center one fully connected to its four nearest neighbors. It is implied that all data paths originate from a single output port and terminate at a single input port.

It can be seen from Figures 4 and 6 that the communication registers are connected differently than the general-purpose registers. The ROUTER, NORTH, and SOUTH registers behave as previously described for general communication, and they provide word-wide input and output ports. The EAST and WEST registers, on the other hand, only behave traditionally when stored to; when loaded from, their data is modified before reaching the logic units. Also, they only have 1-bit input and output ports; the EAST output port consists of the EAST register's most significant bit ("msb"), and the WEST output port consists of the WEST register's least significant bit ("lsb"). The EAST and WEST registers will be discussed in greater detail with the Shift Control subfield.

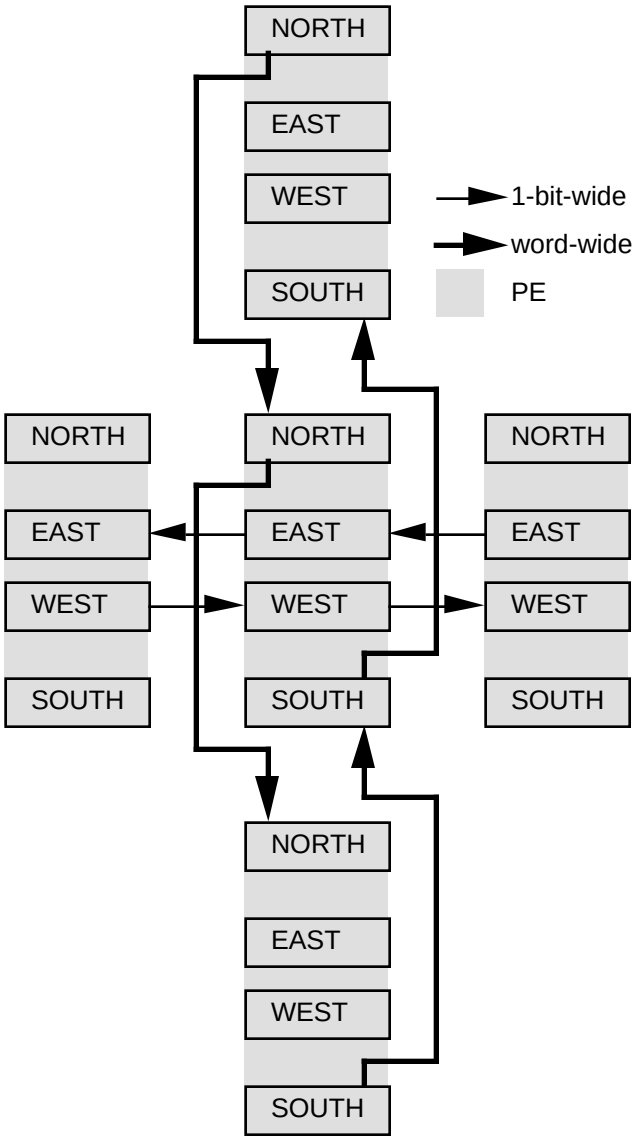


Figure 6. Communication Register Connection

As indicated by Figure 4, the CONDITION register is not a selectable register, nor even readable by the PE; it is only directly accessible outside the PE by the controller. The structure of the CONDITION register is depicted in Figure 7.

The meanings of the condition flags is relatively straightforward. Flags with designations beginning with "Register" indicate the state of

the last selectable destination register used by the PE; flags beginning with "OUT", "IN2", and "IN1" indicate the respective states of the main registers. Thus the controller can monitor the results produced by the logic units as well as the data transferred from memory to registers. Flags with designations ending with "MSB" mirror the contents of the most significant bit position of their respective registers, and are thus useful in observing the signs of signed values and the results of comparisons, as well as examining other bits when coupled with shifting or rotating. Flags ending with "Zero" use a 1 to indicate when all of the bit positions in their respective registers are 0, and are also useful to monitor the results of comparisons.

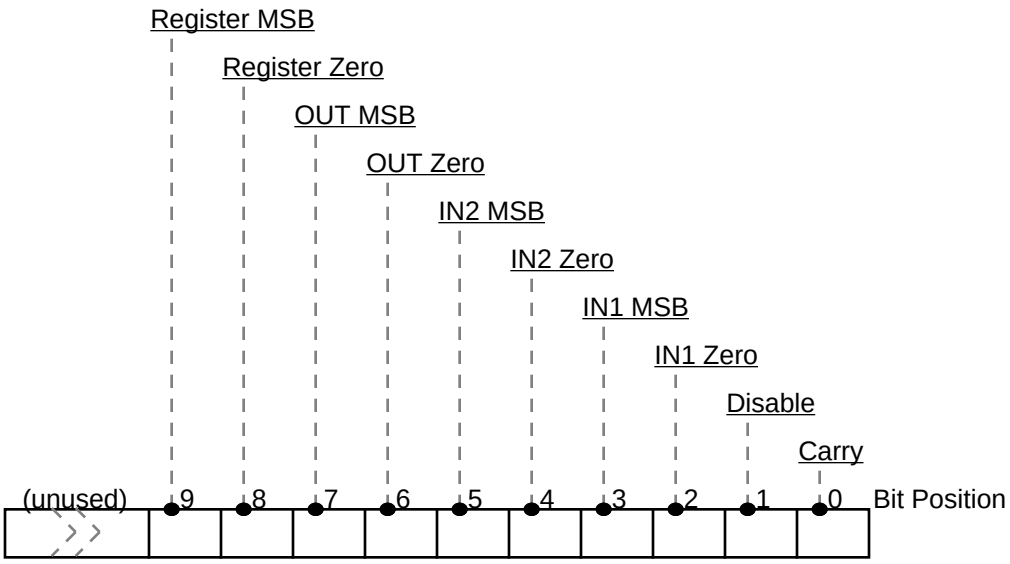
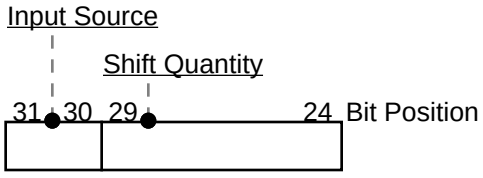


Figure 7. CONDITION Register Format

The Carry flag indicates that a carry-out was generated by the last carry-word computation performed, and the Disable flag indicates that the logical value of the DISABLE register is 1 (that is, the

numerical value is not 0). The DISABLE register operates like a general-purpose register, but its contents have a special effect on the operation of the PE. The Carry flag is discussed with the Carry Control subfield, and the Disable flag and DISABLE register are discussed later with the Disable Control Field of the microinstruction.

Shifting function and specification. The second subfield, Shift Control, actually contains two subfields of its own: Input Source and Shift Quantity. This is shown in Figure 8.



Input Source	
Code	Meaning
00	Constant 0
01	Constant 1
10	Input Port
11	Output Port

Figure 8. Shift Control Subfield Format

Shift Quantity contains the number of bit positions by which the data from the EAST or WEST registers will be shifted before it reaches the logic units. From Figures 4 and 6, it can be seen that the data paths coming from the EAST and WEST registers and going to the main logic units pass through special-purpose logic units, namely variable shift units such as barrel-shifters. The Shift Quantity affects only these shifters, and so has no effect if neither EAST nor WEST is selected as the source register. The shifters can shift between 0 and W bit positions (inclusive) at one time; the EAST shifter shifts left

(westward, or lsb-to-msb), and the WEST shifter shifts right (eastward, or msb-to-lsb).

All the bit spaces shifted in assume the same value, the one specified by the Input Source subfield. As can be seen from Figure 8, the possible input sources consist of the constants 0 and 1, and the register's own input and output ports. If one of the constants is made the source, that value will be shifted-in the number of times specified by the Shift Quantity. If the register's input port is made the source, then by sequentially performing W 1-bit shifts, a PE can transfer an entire data word to its east neighbor and from its west neighbor, or vice versa. If the register's own output port is made the source, a 1-bit shift will produce a 1-bit rotate. Note that in the latter two cases, a Shift Quantity greater than 1 does not result in a complete data transfer because all shifted-in bit spaces will assume the same value as the first. Also of note is the fact that a Shift Quantity of 0 causes the shiftable registers to behave no differently than general-purpose registers, i.e., no communication occurs.

The abilities to shift and rotate are important not only as basic logic functions, but also as means of achieving parallelism on the algorithmic level; this will be seen later. Only allowing a single register the ability to shift a single direction is a natural extension of the view of a word-wide PE as a collection of 1-bit-serial PEs. However, it also provides a convenient basis for a very simple yet efficient design. The concept was retained in order to investigate its usefulness.

To shift all of the bit positions in a 32-bit-wide data word, the Shift Quantity subfield only needs 5 bits. However, if W is 64, 6 bits

are needed, and for reasons discussed later concerning the flexibility of the model, this design has 6 bits and simply ignores one. In a real implementation, this should not be the case.

Logic function and specification. The Register and OUT Operator subfields contain coded representations of the Boolean functions that are to be performed by the main logic units on the three register values. The logic units are simply banks of W "1-of-8" multiplexers. For any value of an index "i" between 1 and W inclusive, the ith multiplexer uses the ith bit from each of the main input registers IN1 and IN2 and the source register specified by the microinstruction, to output one of the eight bits of the operator. Thus the operator is nothing more than the result column of a truth table for the desired Boolean function of (up to) three variables. Additionally, the Register MSB, Register Zero, OUT MSB, and OUT Zero condition flags are set in the CONDITION register when the outputs of the Boolean logic units become available.

As an example, suppose the IN2 register had previously been loaded from memory, and it was now desired to store it in one of the selectable destination registers, and furthermore to compute the exclusive-or ("^") of IN1 and one of the selectable source registers, and store that result to the OUT register for later transfer to memory. The truth table for the desired functions, which automatically yields the desired operators, would be constructed as shown in Table II.

Continuing the example, assume bits 31 of IN1, IN2, and the source contained 1, 0, and 1, respectively. One of the 32 multiplexers forming the selectable-destination logic unit would receive 1, 0, and 1 on its control lines and 11001100 (the bit sequence shown in the "Function:

IN2" column of Table II) on its input. 101 has a decimal value of 5, so the multiplexer would latch bit 5 of its input, which is a 0, to its output, which is bit 31 of the word going to the selectable destination register. Notice that since bit 31 of IN2 is 0 and bit 31 of the result is 0, the Boolean identity function IN2 is demonstrated. Similarly, the input to the OUT destination logic unit is 01011010, and the multiplexer having 1, 0, and 1 on its control lines will latch bit 5 of the input, a 0, to its output, bit 31 of the value going to the OUT register. Notice similarly that since bit 31 of IN1 is 1, bit 31 of the source register is 1, and bit 31 of the result is 0, the Boolean exclusive-or function IN1 ^ SRC is demonstrated. This concludes the example.

Table II

Boolean Operator Generation

Bit Number	IN1 Bit	IN2 Bit	Source Bit	Function: IN2	Function: IN1 ^ SRC
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	1	0
3	0	1	1	1	1
4	1	0	0	0	1
5	1	0	1	0	0
6	1	1	0	1	1
7	1	1	1	1	0
				Selectable Destination Operator	OUT Destination Operator

Given that exactly eight bits are required to describe any Boolean function of up to three variables, it is clear that there are exactly

256 such functions. Of those, at least 39 are interesting enough to merit names. The exact nature of the functions can be seen in the parameter section of any configuration module program, all of which appear in the Appendix.

Also, while the particular order of the bits or the sequence of the registers is not important to the creation of a truth table, those characteristics must necessarily be fixed in the design, and so care must be taken to generate operators with those characteristics. It may be of interest to note that the bit positions and register sequence depicted in Table II are those that were selected for the design that was modeled.

Carry-word computation and specification. The architecture has been provided the ability to perform a full addition or subtraction in two microinstruction cycles. The method involves computing a carry-word with the two operands in the first cycle, and then performing the Boolean addition of the operands and the carry-word in the second cycle.

The Boolean addition of three operands is a straightforward application of an easily-generated operator to the Boolean logic units. The computation of the carry-word is achieved with a special-purpose logic unit -- a carry-lookahead unit. To keep the operation and implementation of the PE as simple as possible, the carry-lookahead unit has inputs fixed to the main input registers IN1 and IN2, and uses the same output path as the selectable-destination Boolean logic unit. Therefore, the (selectable-destination) carry-word computation and the selectable-destination Boolean operation are mutually exclusive. This can be observed in Figure 4. Alternatively, the output of the carry-lookahead unit could be designed to be fixed to a particular register,

call it the CARRY register, in order that the selectable-destination Boolean operation not be sacrificed. However, since the CARRY register would have to be selectable as a source, the number of general-purpose registers would be reduced to nine. Since it is unclear which if either scheme is better, the choice was made arbitrarily with the intent of studying its effect.

The flexibility designed into the carry-lookahead unit is one of the powerful features of this architecture. The operation of the unit is governed by the Carry Control field shown in Figure 5. It consists of 6 1-bit subfields: Use IN1, Use IN2, Invert IN1, Invert IN2, Use Absolute Carry-In, and Carry-In Value. This format is depicted in Figure 9.

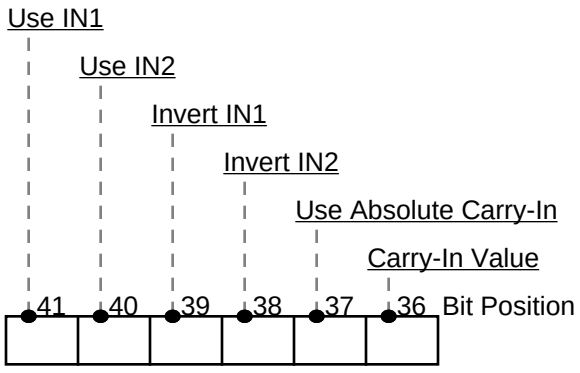


Figure 9. Carry Control Subfield Format

Setting the Use IN1 bit causes the carry-lookahead unit to use the value of the main input register IN1 in the computation of the carry-word; Use IN2 functions similarly. Having separate controls for IN1 and IN2 provides part of the means to perform simple and efficient increment operations. It also provides the means to determine which output to

enable and which to disable; if neither Use IN1 nor Use IN2 is set, the Boolean logic unit output is enabled, and the carry-lookahead unit output is disabled. Setting Invert IN1 causes the carry-lookahead unit to use the inverse of the value of IN1 in the computation of the carry-word, if and only if its use was specified by the setting of Use IN1; Invert IN2 functions similarly. This provides the means to perform simple and efficient subtraction and negation operations.

Setting Use Absolute Carry-In causes the carry-lookahead unit to obtain the value for the carry-in from the adjacent Carry-In Value bit of the microinstruction; otherwise, it is obtained from the Carry flag of the CONDITION register, which can be seen in Figure 7. This provides the means to perform multi-precision arithmetic, as well as the increment operations mentioned earlier. The Carry flag is set by the computation of the carry-word; when a carry-word is not being computed, the Carry flag is not affected. This allows the controller the freedom to not perform the subsequent Boolean addition or subtraction operation immediately following the carry-word computation, which can be useful in obtaining maximum economy of cycles.

There are 64 combinations of specifications for the Carry Micro-Operation Field, of which 23 are distinct. These are listed, usefulness aside, along with the algebraic significance of the carry-words generated, in Table III.

Note that a single-register increment can be specified, but a single-register decrement cannot. However, a decrement on a single register is possible if the other register can be made to contain 0. In that case, specifying a zero carry-in value causes the 1s-complement of 0 to be added instead of the 2s-complement, i.e., —1 in 2s-complement.

Table III

Carry-Word Computation Specifications

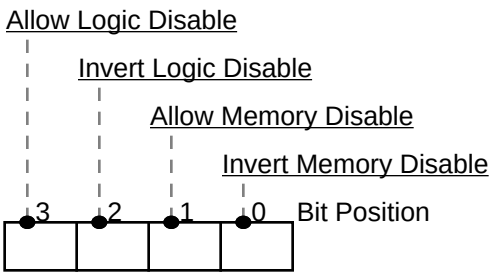
Use IN1	Use IN2	Invert IN1	Invert IN2	Use Carry-In Value	Carry-In Value	Algebraic Significance
0	0	x	x	x	x	no operation
0	1	x	0	0	x	IN2 + C
0	1	x	0	1	1	IN2 + 1
0	1	x	1	0	x	- IN2 - 1 + C
0	1	x	1	1	0	- IN2 - 1
0	1	x	1	1	1	- IN2
1	0	0	x	0	x	IN1 + C
1	0	0	x	1	1	IN1 + 1
1	0	1	x	0	x	- IN1 - 1 + C
1	0	1	x	1	0	- IN1 - 1
1	0	1	x	1	1	- IN1
1	1	0	0	0	x	IN1 + IN2 + C
1	1	0	0	1	0	IN1 + IN2
1	1	0	0	1	1	IN1 + IN2 + 1
1	1	0	1	0	x	IN1 - IN2 - 1 + C
1	1	0	1	1	0	IN1 - IN2 - 1
1	1	0	1	1	1	IN1 - IN2
1	1	1	0	0	x	IN2 - IN1 - 1 + C
1	1	1	0	1	0	IN2 - IN1 - 1
1	1	1	0	1	1	IN2 - IN1
1	1	1	1	0	x	- IN1 - IN2 - 2 + C
1	1	1	1	1	0	- IN1 - IN2 - 2
1	1	1	1	1	1	- IN1 - IN2 - 1

Disable Architecture

The last feature of this architecture to be discussed is the ability to selectively disable individual PEs based on their state. For even greater flexibility, this control is provided to both the PE and the controller. Such a capability is outside of the strict definition of SIMD, but is important in providing the flexibility necessary to

implement many useful forms of algorithmic parallelism. Take for example the iterative process of value normalization required for floating-point addition and subtraction operations. The values contained in different PEs are more than likely to require a different number of iterations to reach a normalized state. Allowing some PEs to perform more iterations than others means prohibiting some PEs from performing unwanted iterations, and one way to accomplish that is to disable those PEs.

The architecture of the disable control system consists of two components: the logical value of the DISABLE register that appears in Figure 4, which is accessible by the PE directly and by the controller through the Disable condition flag of the CONDITION register, and the control signals embedded in the Disable Control field of the microinstruction. The field contains four signals, and these are depicted in Figure 10.



ALD & (ILD ^ DIS) = Disable Logic Operation

AMD & (IMD ^ DIS) = Disable Memory Operation

Figure 10. Disable Control Field Format

Figure 10 also shows the Boolean equations for the disable functions, which operate as follows. If the Allow Logic Disable bit is

set, and either the DISABLE register contains a 1 and the Invert Logic Disable bit is 0, or the DISABLE register is 0 and the Invert Logic Disable bit is 1, then the outputs of the two Boolean logic units and the carry-lookahead unit will be disabled, as well as the setting of the two Register, the two OUT, and the Carry condition flags in the CONDITION register. Similarly, if the Allow Memory Disable bit is set, and either the DISABLE register contains a 1 and the Invert Memory Disable bit is 0, or the DISABLE register is 0 and the Invert Memory Disable bit is 1, then the latching of the memory bus into the IN1 and IN2 registers and the memory is disabled, as well as the setting of the two IN1 and the two IN2 condition flags in the CONDITION register.

The significance of these functions lies in the cooperative nature of the disable control between the controller and the PE itself. In effect, the controller allows disabling to be possible, and then the controller and the PE cooperate to determine whether or not the PE should be disabled. This is a very flexible and powerful feature, and makes a variety of parallelization schemes possible. Some of these can be seen in the PE configuration experiments presented in the section following this one, "Arithmetic Performance".

Behavioral Model

The PE and configurations are modeled in the Verilog HDL partly because of its availability, partly because it was recommended, and partly because of its purported resemblance to the already-familiar C programming language. The choice turned out to be satisfactory, especially for modeling parallel tasks and allowing precise accounting of clock cycles. A good description of Verilog appears in the introduction of the Verilog-XL Reference Manual [23].

> The Verilog Hardware Description Language ("HDL") describes a hardware design or part of a design. Descriptions of designs in the Verilog HDL are Verilog models. The Verilog HDL is both a behavioral and structural language. Models in the Verilog HDL can describe both the function of a design and the components and connections to the components in a design.
>
> ....
>
> The basic building block of the Verilog HDL is the module. The module format facilitates top-down and bottom-up design. A module contains a model of a design or part of a design. Modules can incorporate other modules to establish a model hierarchy that describes how parts of a design are incorporated in an entire design. The constructs of the Verilog HDL, such as its declarations and statements, are enclosed in modules.
>
> The Verilog HDL behavioral language is structured and procedural like the C programming language. .... The behavioral language provides the following capabilities:
>
> - structured procedures for sequential or concurrent execution
> - explicit control of the time of procedure activation specified by both "delay" expressions and by value changes called "event" expressions
> - explicitly named events to trigger the enabling and disabling of actions in other procedures
> - procedural constructs for conditional, if-else, case, and looping operations
> - procedures called "tasks" that can have parameters and non-zero time duration
> - procedures called "functions" that allow the definition of new operators
> - arithmetic, logical, bit-wise, and reduction operators for expressions

Further information about Verilog can be found in [23]; unfortunately, no adequate tutorials nor better references can be recommended at this time.

The model of the PE is behavioral, and it models the PE exactly as previously described. It evolved through at least five versions, the last one of which is named "pe4". It resides in the UNIX Verilog file

"pe.v" which is reproduced in the Appendix, and pertinent sections appear below with the text.

Module "pe4" consists of a single module with no submodules. The first statement is the module declaration, and it contains a list of the input and output port declarations.

```
module pe4
  (p_clock, p_reset, p_instr, p_flags,
   n_in, s_in, e_in, w_in, r_in,
   n_out, s_out, e_out, w_out, r_out,
   m_clock, m_write, m_addr, m_in, m_out,
   dump_reg, dump_mem);
```

This statement simply declares that the ports exist; it does not define what they are. The port definition statements normally follow immediately after the declaration, but in this case some of the definitions are dependent upon parameters which must be defined first.

```
parameter // Defaults, should be overridden by instantiating module:
  ADDR_WIDTH  = 10,          //  width of PE memory address, in bits
  WORD_WIDTH  = 32,          //  width of PE data word, in bits
  MEM_LENGTH  = 1024,        //  length of PE memory, in words
  PE_NAME     = "undefined"; //  name of PE for dump identification
```

It must be mentioned that two methods of interfacing to the module are used: direct object access, and ports. All data objects in an instance of a module can be accessed directly by the instantiating module, so ports are technically not required. However, ports are used in this module to support better programming style, since the PE design has them. The parameters defined above describe special features of the module that are not features of the PE itself but that provide great flexibility. For example, while having a single data object that defines the PE's data word width is convenient, having the ability for the instantiating module to arbitrarily set it is much more so. Additionally, it is distinctly simpler to access data directly, and with features that are not part of the PE design, there is no reason not to

do so. Since the values of the parameters can be set externally, the ones used in the PE module merely serve as defaults and examples. Note that the MEM_LENGTH parameter is an unfortunate necessity; surprisingly, Verilog has no exponent, root, or logarithm functions, and so there is no way to calculate the length of the PE memory from the width of its address, which in the default case here would be 2 to the power of 10.

With the special parameters defined, the input and output ports can then be defined.

```
input // (1 bit wide)
  p_clock, p_reset,     //  PE clock & reset, rising-edge triggered
  e_in, w_in,           //  east (lsb) & west (msb) communication
  m_clock,              //  PE memory clock for external access, rising-edge
  m_write,              //  PE memory read/write control for external access:
                        //    read = 0, write = 1
  dump_reg, dump_mem;   //  PE register & memory dump clocks, rising-edge
input  [ADDR_WIDTH-1:0]
  m_addr;               //  PE memory address for external access
input  [WORD_WIDTH-1:0]
  n_in, s_in, r_in,     //  PE north & south & router (word) communication
  m_in;                 //  PE memory data for external access
input  [ADDR_WIDTH+46-1:0]
  p_instr;              //  PE microinstruction (described below)
output // (1-bit wide)
  e_out, w_out;         //  east (msb) & west (lsb) communication
output [9:0]
  p_flags;              //  PE condition flags (described below)
output [WORD_WIDTH-1:0]
  n_out, s_out, r_out,  //  PE north & south & router (word) communication
  m_out;                //  PE memory data for external access
```

The ports are exactly as described previously for the design of the PE as shown in Figure 4, with a few additions. An interface to the PE memory is incorporated into the model so that the controller can access it; these ports are designated "m_...". However, since loading and reading PE memory by the controller has no effect on the execution of arithmetic operations, and since direct data access is notably simpler than using ports, it was decided to access memory directly and ignore the existence of the memory ports. For this reason, the memory access operation will not be treated further.

Also, a number of clock signals are defined. These clocks are all rising-edge-active, that is, whenever the signal changes from low (0) to high (1), some set of actions is performed once and is completed before the next such transition. The activities they control are also all independent, so activating them all simultaneously would not affect PE performance, although possible resource contention might produce unpredictable results. The main PE clock "p_clock" governs all the PE operation described previously for the design. The reset clock "p_reset" clears all internal registers; this is needed because Verilog initializes registers to an "unknown" state, but no such state exists in the PE design. The "dump_mem" and "dump_reg" clocks cause the PE to disgorge the contents of its memory and/or registers to the standard output (hence the usefulness of the PE_NAME parameter); Verilog of course can do this, but not nearly as conveniently nor with as useful a format as custom routines. Note that the reset and dump operations do not warrant consideration as part of the PE design.

With the ports defined, the data objects that they will be connected to can be declared and defined, along with other objects used internally by the PE.

```
reg do_mem_op; // (1 bit wide)
reg [7:0] reg_op, out_op;
reg [9:0] p_flags;
reg [ADDR_WIDTH-1:0] r;
reg [WORD_WIDTH-1:0] mem[0:MEM_LENGTH-1],
                     regs[0:15],
                     in_1, in_2, out, m_out,
                     mem_wrk, alu_wrk;
reg [WORD_WIDTH:0] car_wrk;
wire e_out, w_out; // (1 bit wide)
wire [WORD_WIDTH-1:0] re, rw;
integer i, j, k, reg_dump_ct, mem_dump_ct, clock_ct;
```

The significance of these objects will become clear. Normally, the next step would be to assign registers to the ports. In this case,

however, the assignments depend on a few parameters and constants that must be defined first.

```
//  PE microinstruction field definitions and content parameters:
`define   mb_addr p_instr[ADDR_WIDTH+46-1:46]  //  memory bus transfer address
`define   mb_srce p_instr[45]  //  mem bus transfer source is mem (1) or OUT (0)
`define  mb_d_mem p_instr[44]  //  memory bus transfer dest is memory, 1=true
`define  mb_d_in2 p_instr[43]  //  memory bus transfer dest is IN2, 1=true
`define  mb_d_in1 p_instr[42]  //  memory bus transfer dest is IN1, 1=true
  //  Note:  A carry-word is computed using registers IN1 and/or IN2 and a
  //  specified carry-in bit.  A carry operation is implied by specifying the
  //  registers to use, and since the carry-word is placed into the destination
  //  register, that ALU is disabled.  The carry-out is placed in the carry
  //  flag of the p_flags register, which remains unchanged by ALU operations.
`define   car_in1 p_instr[41]  //  use IN1 in carry-word computation, 1=true
`define   car_in2 p_instr[40]  //  use IN2 in carry-word computation, 1=true
`define  car_nin1 p_instr[39]  //  use inverse of IN1 (if used), 1=true
`define  car_nin2 p_instr[38]  //  use inverse of IN2 (if used), 1=true
`define  car_srce p_instr[37]  //  use value for carry-in (or carry flag) 1=true
`define   car_val p_instr[36]  //  carry-in value (if used), 1 or 0
`define  srce_reg p_instr[35:32]  //  source register of ALU operation...
`define   ew_srce p_instr[31:30]  //  source of east or west shifted-in bit...
  //  mnemonics for ew_srce values:
  parameter EW0  = 2'b00,  //  shift constant 0 into east or west register
            EW1  = 2'b01,  //  shift constant 1 into east or west register
            EWIN = 2'b10,  //  shift input port bit into east or west register
            EWOU = 2'b11;  //  shift output port bit into east or west register
`define     shift p_instr[29:24]  //  shift east or west source reg "shift" bits
`define     regop p_instr[23:16]  //  ALU operand for dest register result
`define  dest_reg p_instr[15:12]  //  destination register of ALU operation...
`define     outop p_instr[11:4]   //  ALU operand for OUT register result
  //  mnemonics for srce_reg and dest_reg values:
  parameter R0 = 4'b0000, R1 = 4'b0001, R2 = 4'b0010, R3 = 4'b0011,
            R4 = 4'b0100, R5 = 4'b0101, R6 = 4'b0110, R7 = 4'b0111,
            R8 = 4'b1000, R9 = 4'b1001, RD = 4'b1010, RR = 4'b1011,
            RN = 4'b1100, RS = 4'b1101, RE = 4'b1110, RW = 4'b1111;
`define   alu_dis p_instr[3]  //  allow disabling of ALU operation, 1=true
`define alu_dis_i p_instr[2]  //  invert PE disable bit for ALU op, 1=true
`define    mb_dis p_instr[1]  //  allow disabling of memory bus operation....
`define  mb_dis_i p_instr[0]  //  invert PE disable bit for memory bus op....

//  PE condition flag definitions and condition register bit positions:
//    Note:  the carry flag is only affected by a carry operation.
`define reg_msb_f p_flags[9]  //  m.s.b. of destination register
`define reg_zer_f p_flags[8]  //  if destination register == 0
`define out_msb_f p_flags[7]  //  m.s.b. of register OUT
`define out_zer_f p_flags[6]  //  if register OUT == 0
`define in2_msb_f p_flags[5]  //  m.s.b. of register IN_2
`define in2_zer_f p_flags[4]  //  if register IN2 == 0
`define in1_msb_f p_flags[3]  //  m.s.b. of register IN_1
`define in1_zer_f p_flags[2]  //  if register IN1 == 0
`define disable_f p_flags[1]  //  state of PE disable bit
`define   carry_f p_flags[0]  //  carry-out of last carry operation
```

These are all of the microinstruction fields and condition flags previously discussed for the PE design. The only exception is the addition of mnemonics for those fields both containing encoded information (as opposed to straight numerical values) and consisting of

more than a single bit, namely the Shift Control Input Source (ew_srce), Source Register (srce_reg), and Destination Register (dest_reg) fields. These are handy not only in the PE module program, but also for the module programs that use instances of PEs.

Now that the desired register mnemonics are defined, the registers can be attached to the ports.

```
assign n_out = regs[RN],
       s_out = regs[RS],
       re    = regs[RE],  //  need since cannot access bits of regs[i]!!!
       e_out = re[WORD_WIDTH-1],
       rw    = regs[RW],  //  need since cannot access bits of regs[i]!!!
       w_out = rw[0],
       r_out = regs[RR];
```

The only assignments that bear mention are those of the EAST and WEST registers (regs[RE] and regs[RW]). The EAST register output port (e_out) is supposed to be connected to the msb of regs[RE]. Unfortunately, Verilog does not provide a means to access the individual bits of a member of a register array. It does allow the ith register member of a register array to be accessed with the notation "reg_array[i]", and it does allow the ith bit of a non-arrayed register to be accessed with "plain_reg[i]", but it does not allow the jth bit of the ith register member of a register array to be accessed (as might reasonably be expected) with "reg_array[i][j]". The only option is to hard-wire (assign) regs[RE] to an intermediate, non-arrayed register (re), and then assign the msb of that (re[WORD_WIDTH-1]) to e_out. The same problem occurs with regs[RW].

With all the declarations, definitions, and port assignments completed, the "operational" program blocks can be entered. There are six program blocks, and they can be seen in the Appendix: the "initial" block, the reset cycle block, the microinstruction cycle block, the memory access cycle block, the register dump cycle block, and the memory

dump cycle block. The initial block runs once, as soon as Verilog begins executing the module, and simply resets the cycle counter variables. The other five blocks are "always @ (posedge <clock>)" blocks corresponding to the five clock signals previously described. The only one pertinent to this discussion is the microinstruction cycle block, and it is presented below.

The microinstruction cycle block is divided into four sections: initial memory transfer operation, logic/carry operation, final memory transfer operation, and exit operation. The initial memory transfer operation runs only if memory operations are not disabled as previously described, and consists of "latching the data onto the bus", or copying the contents of the specified data transfer source to the memory "working" register mem_wrk.

```
// Microinstruction cycle program
always @ (posedge p_clock)
  begin

    // Memory operation source-to-bus section
    if (~(`mb_dis & (`mb_dis_i ^ (regs[RD] || 0)))) // not disabled
      begin
        do_mem_op = 1; // determine this now in case regs[RD] changes
        if (`mb_srce) mem_wrk = mem[`mb_addr]; //  `mb_srce == memory
        else          mem_wrk = out;           //  `mb_srce == OUT register
      end
    else do_mem_op = 0;
```

This must be done before the OUT register is modified by the logic operation, as discussed previously for the PE design. Also, whether or not the memory operations are disabled must be determined in case the contents of the DISABLE register are changed by the logic operation before the memory bus-to-destination operation. Note that a logic operation must be performed on the DISABLE register to obtain its logic (as opposed to numeric) value since Verilog provides no explicit means of doing so.

The next section is the logic/carry operation, and only runs if logic/carry operations have not been disabled. It is divided into two parts: the setup, which follows, and the execution.

```
// ALU/carry operation section
if (~(`alu_dis & (`alu_dis_i ^ (regs[RD] || 0))))  //  not disabled
  begin

    //  Load data sources to set up for operation
    case (`srce_reg)
      RR: alu_wrk = r_in;
      RN: alu_wrk = n_in;
      RS: alu_wrk = s_in;
      RE:
        begin
          alu_wrk = regs[RE] << `shift % WORD_WIDTH;  //  zeros shifted in
          if (   `ew_srce == EW1
              || `ew_srce == EWIN && e_in         //  if source is a 1,
              || `ew_srce == EWOU && e_out )      //    then "shift" it in
            for (i=0; i<`shift%WORD_WIDTH; i=i+1) alu_wrk[i] = 1;
            //  use loop since Verilog doesn't allow variable part-select!!!
        end
      RW:
        begin
          alu_wrk = regs[RW] >> `shift % WORD_WIDTH;  //  zeros shifted in
          if (   `ew_srce == EW1
              || `ew_srce == EWIN && w_in
              || `ew_srce == EWOU && w_out )      //  if source is a 1,
            for (i=0; i<`shift%WORD_WIDTH; i=i+1)  //    then "shift" it in
              alu_wrk[WORD_WIDTH-1-i] = 1;
            //  use loop since Verilog doesn't allow variable part-select!!!
        end
      default: alu_wrk = regs[`srce_reg];
    endcase
    if (`car_srce) car_wrk = `car_val;
    else           car_wrk = `carry_f;
    reg_op = `regop;  //  needed since cannot access bits of `regop!!!
    out_op = `outop;  //  needed since cannot access bits of `outop!!!
```

This is where the four logic/carry working registers are set up: alu_wrk, car_wrk, reg_op, and out_op. The content of the specified selectable-source register is copied into the working source register alu_wrk; this means if the source is a communication register, the value at the port is copied instead of the register, and if the register is shiftable, its value is shifted and the specified source bit "shifted" in. Unfortunately, Verilog provides no convenient means to shift-in 1s; it only shifts-in 0s, and if 1s are required, they must be set manually. The problem is exacerbated by the equally unfortunate fact that Verilog

does not provide a means to access a variable sequence of bits inside of a register. It does allow a variably-chosen ith bit to be accessed with the notation "plain_reg[i]", and it does allow a constantly-defined bit sequence to be accessed with "plain_reg[31:24]", but it does not allow a variably-chosen bit sequence to be accessed (as might reasonably be expected) with "plain_reg[i:j]". Therefore, a "for" loop must be used to set the shifted-in bits, and in the interest of model execution speed, the loop only sets bits to 1 and is executed only if 1s are required.

Next, the specified carry-in value is copied into the W-plus-1-bit-wide carry-word working register car_wrk. Finally, the Register and OUT Operator fields of the microinstruction, `regop and `outop, are copied into working registers reg_op and out_op. This is necessary because `regop and `outop are syntactically equivalent to "p_instr[23:16]" and "p_instr[11:4]", and this syntax prohibits direct access of individual bits.

Now that the working registers are set up, execution of the logic/carry operation can take place.

```
// Execute operation and store results
if (`car_in1 || `car_in2) // carry operation
  begin
    for (i=0; i<WORD_WIDTH; i=i+1)
      begin
        out[i]      = out_op[{in_1[i], in_2[i], alu_wrk[i]}];
        car_wrk[i+1] =  (in_1[i] ^ `car_nin1) & `car_in1 & car_wrk[i]
                      | (in_2[i] ^ `car_nin2) & `car_in2 & car_wrk[i]
                      | (in_1[i] ^ `car_nin1) & `car_in1 &
                        (in_2[i] ^ `car_nin2) & `car_in2;
      end
    regs[`dest_reg] = car_wrk;
    `carry_f   = car_wrk[WORD_WIDTH];
    `reg_zer_f = !car_wrk[WORD_WIDTH-1:0];
    `reg_msb_f = car_wrk[WORD_WIDTH-1];
  end
else                        // ALU operation
  begin
    for (i=0; i<WORD_WIDTH; i=i+1)
      begin
        out[i]     = out_op[{in_1[i], in_2[i], alu_wrk[i]}];
```

```
            alu_wrk[i] = reg_op[{in_1[i], in_2[i], alu_wrk[i]}];
          end
        regs[`dest_reg] = alu_wrk;
        `reg_zer_f = !alu_wrk[WORD_WIDTH-1:0];
        `reg_msb_f = alu_wrk[WORD_WIDTH-1];
      end
    `out_zer_f = !out;
    `out_msb_f = out[WORD_WIDTH-1];
    `disable_f = regs[RD] || 0;  //  ok here because wont change elsewhere

  end  //  of ALU/carry operation section
```

Separate program sequences for operations including and excluding a carry-word computation were created to enhance understandability, although they are fairly similar. In both cases, a loop is used to perform the actual computation, since the Verilog behavioral language has nothing to directly describe a parallel multiplexer or carry-lookahead circuit. The loop iterates over each bit position of the data word. The statements performing the Boolean functions exactly mimic the individual 1-of-8 multiplexers described previously. Each bit of the OUT-destination result goes directly into the OUT register since it is non-arrayed and its bits are individually accessible. Each bit of the selectable-destination result, however, goes back into the alu_wrk register (where the selectable-source value was) for later transfer to the selectable-destination register since it is a member of a register array and its bits are not directly accessible. For the non-carry-operation case, calculation of an OUT-destination bit precedes calculation of the corresponding selectable-destination bit because the former depends on the latter and the latter modifies itself.

The carry-word computation statement presented here in the Verilog behavioral language is simply an expression of the well-known carry-lookahead generate-propagate formula,

carry[i+1] = generate[i] | (propagate[i] & carry[i]) ,

where

```
generate[i]  = operand_1[i] & operand_2[i]
```

and

```
propagate[i] = operand_1[i] | operand_2[i]  .
```

Substitution and reduction yields

```
carry[i+1]   = (operand_1[i] & operand_2[i])
               | (operand_1[i] & carry[i])
               | (operand_2[i] & carry[i])  ,
```

which appears almost verbatim in the program. Minor differences are due to the use of slightly different variable names and expressions for the operands that incorporate the Use INx and Invert INx Carry Control microinstruction signals.

With the computation loop completed, the selectable-destination result is copied to the appropriate register, and the condition flags are set up in the CONDITION register as specified previously for the PE design. There are also minor differences called for in these procedures depending on whether or not the operation includes a carry-word computation. In concluding the description of the logic/carry operation section, it should be mentioned that the Disable condition flag is set here instead of in the exit operation because it cannot be changed either by memory operation or if logic/carry operation is disabled.

The next program section is the final memory operation. It runs only if memory operations are not disabled, and consists of "latching the data from the bus", or copying the contents of mem_wrk to the specified data transfer destination(s).

```
//  Memory operation bus-to-destination section
if (do_mem_op)  //  mem ops not disabled
  begin
    //  only latch TO memory if not latched FROM memory
    if (`mb_d_mem && !`mb_srce) mem[`mb_addr] = mem_wrk;
    if (`mb_d_in2)
      begin
        in_2      = mem_wrk;
        `in2_zer_f = !in_2;
```

```
        `in2_msb_f = in_2[WORD_WIDTH-1];
      end
    if (`mb_d_in1)
      begin
        n_1      = mem_wrk;
        `in1_zer_f = !in_1;
        `in1_msb_f = in_1[WORD_WIDTH-1];
      end
  end  //  of memory operation section
```

Also, the condition flags are set up in the CONDITION register as specified previously for the PE design.

Finally, the exit operation must be performed prior to termination of the microinstruction cycle program.

```
    clock_ct = clock_ct + 1;

  end  //  of microinstruction cycle program
```

The cycle counter is not a feature of the PE design but of the model program. It can be used by the instantiating module for direct monitoring, making tracking the performance of various algorithms extremely simple. This concludes the discussion of the PE behavioral model as well as of the PE in general. The following section, "Arithmetic Performance", details the design, modeling, and performance of configurations of PEs for implementing arithmetic operations.

ARITHMETIC PERFORMANCE

Overview

As mentioned previously, one measure of the potential for success of a particular processor architecture is the number of clock and microinstruction cycles required to perform basic operations. In the case of a processor intended as an element in a reconfigurable multiprocessor computer, the measure includes the amount of speedup obtained by the application of multiple processors to a single operation. The complete set of experiments considered for this study consists of the 40 (non-trivial) combinations of: addition/subtraction, multiplication, and division; integer and floating-point; 32- and 64-bit-wide data words; 1, 2, 4, and 8 PEs.

The nature of a PE configuration can be observed from the general MSIMD multiprocessor architecture for which the PEs were intended. This appears in Figure 11.

The PEs are shown here connected in a standard two-dimensional rectangular mesh, but they can also be connected in a three-dimensional spiral or circular cylinder or torus without impacting this discussion. Also, there is a global data routing system that is not shown which has individual connections to each of the PEs.

The general operation of the system is relatively straightforward. Each of the microprogram memory units ("MMUs") that can be seen in Figure 11 preceding the rows of PEs contains all of the microinstruction sequences required to perform the machine operations. The MMUs receive a clock signal and a starting address broadcast from the microcontrol

unit ("MCU"), and they in turn broadcast the clock signal and a microinstruction to all of the PEs in their row. The programming of the MCU is designed to synchronize with and control the flow of the microinstructions from the MMUs.

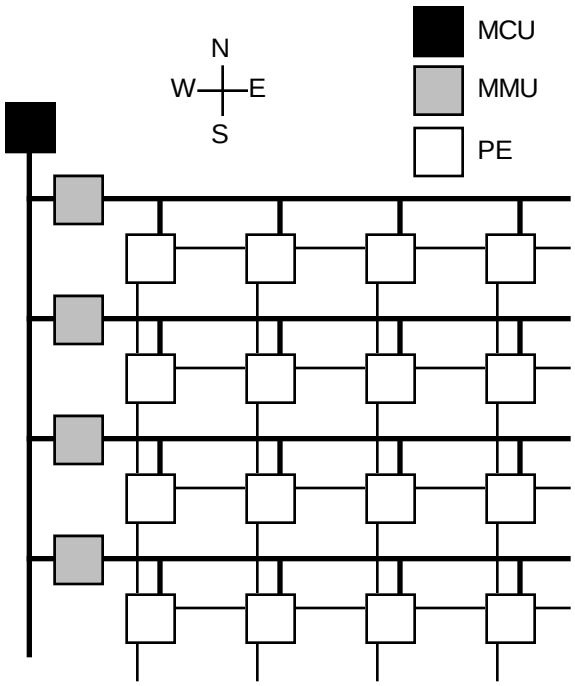


Figure 11. Target MSIMD Multiprocessor Architecture

For purely SIMD operations, the programs contained within the MMUs are identical. To apply parallelism to individual operations, the MSIMD operation is invoked by defining parallel groups of PEs and designing microprograms such that the individual elements (of a group) receive different microinstructions. As can be seen in Figure 11, the only way for PEs to receive different microinstructions is for them to be on different rows. To facilitate inter-PE communication, they should use word-wide data paths, and so they should also be adjacent in the North

and South directions. Therefore, a parallel group consists of a segment of a column. The partitioning of the PE space into column segments, called "configuring", is entirely a function of the programs placed into the MMUs. As such, the "configurations" can be different from operation to operation, making the architecture "reconfigurable" with the intent of changing the "method" used to perform the operation. The SIMD paradigm then applies among parallel groups which represent the executors of machine instructions (the external view), while the PEs inside a group operate in a synchronous-MIMD fashion on the microinstruction scale (the internal view).

To accommodate branching within the operation algorithm, the disable feature of the PE architecture is employed. The PE can be disabled based on its condition by first instructing it to set its DISABLE register to the result of some Boolean operation, and then instructing it to allow disabling on individual microinstructions. The microinstruction stream does not actually branch; all of the optional microinstructions are broadcast in some programmed sequence, and individual PEs "decide" which ones to ignore by disabling themselves. Note that the number of instruction cycles required to perform a branch is equal to the number of branches multiplied by the number of cycles required for the longest branch, plus the overhead of setting up the DISABLE register. Also note that in this architecture, the CONDITION register serves no vital purpose.

Like the PE, the configurations are modeled in the Verilog behavioral language. The typical configuration program consists of a single module that first instantiates and connects the required number of PEs and then executes a sequence of microinstructions on them. Since

a configuration simulation executes linearly and singly within a trial, the entire program is placed within a Verilog "initial" block. Also, since the actual operation of the MCU and MMUs has no bearing on the performance of arithmetic operations, they are not specifically modeled; instead, their functionality is simulated by the capabilities of the Verilog behavioral language itself. This makes writing and understanding the behavioral configuration programs significantly easier because all that is necessary to perform a microinstruction cycle is to set up the p_instr of each PE individually and then toggle all the p_clock lines simultaneously. Each program also contains a large header which includes descriptions of the microinstruction format, fields, registers, flags, and operations, definitions of mnemonic parameters, and general instructions on configuration programming. The mnemonics are especially useful because they form a micro-assembly language. The details of configuration programming can be seen in any configuration module program, all of which appear in the Appendix, but note the floating-point programs are written in a much more advanced style than the integer ones.

Before proceeding further, a few items remain to be mentioned. First, there was significant evolution of the PE design that occurred during the documentation as well as the experimentation processes. Thus it is that the design used in the majority of experiments, module "pe3", is slightly different than the one described previously, "pe4". Also, there is a conceptual difference in the target multiprocessor architecture for which the configuration model programs are developed. These differences and their significance are outlined in the following paragraphs.

PE design: shift control input source. The Shift Control Input Source specification does not exist in the microinstruction of the PE used. The responsibility of connecting a PE's EAST and WEST communication register input ports to its own output ports, its neighbors output ports, or the constants 0 or 1 fall to a controller. Since it does not affect the PE function, it has zero impact on performance, and only a slight impact on the form of the configuration model programs.

PE design: carry-lookahead logic. The carry-lookahead logic unit is not physically located in the PE used; its functionality resides instead in the global data router. This originates from the 1-bit-serial PE design where a sub-router has direct access to the IN1 and IN2 registers and the control signals in the microinstruction in order to implement arbitrary-precision arithmetic (precision-reconfigurability). This sub-router is responsible for supplying the carry-word to the PE's ROUTER input port and setting the PE's Carry condition flag. Since the sub-router itself is not modeled, its carry-word computation function is included in the PE model program, so the PE actually sets its own ROUTER register input port. Since no instance was found where either the ROUTER register was needed for something besides the carry-word, or no free general-purpose register was available, it has zero impact on performance, and only a slight impact on the form of the configuration model programs.

PE design: carry-word computation. A design rule exists in the PE used that is absent from the PE described: a carry-word computation and a non-carry operation cannot be performed in a single microinstruction cycle. The origin of this characteristic is

undetermined, but may be as simple as a miscommunication. Since it might be possible to rewrite the loops in the configuration model programs to take advantage of the additional concurrency afforded by the elimination of the rule, the impact on performance might be significant. However, it is exceedingly unlikely that such performance difference would completely invalidate the results obtained with the PE design used.

Target multiprocessor architecture. In the target multiprocessor architecture for which the configuration model programs are developed, instead of there being one MMU for each row of PEs, each PE incorporates a combined microinstruction program memory and control unit. This is basically Figure 11 with the symbols for the MMUs removed; this is shown in Figure 12.

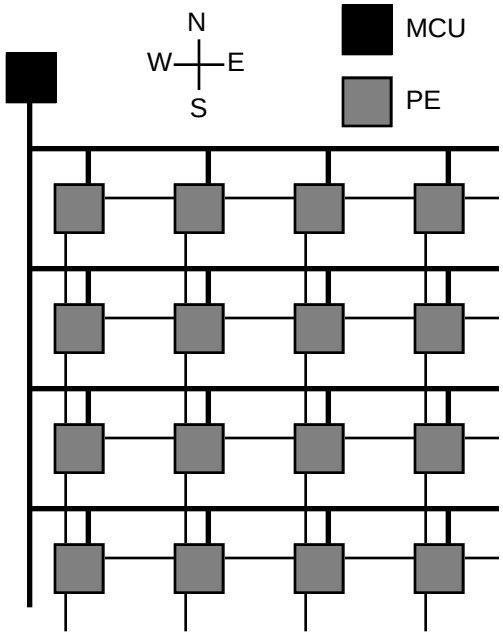


Figure 12. Alternate Multiprocessor Architecture

The MCU broadcasts the starting address of the machine operation program and the clock to all of the PEs internal control units, and it counts the cycles to determine when the operation has been completed. A PE's controller uses the its condition flags and EAST and WEST register outputs to control execution directly, and synchronization of PEs in a parallel group for branch equalization is attained by using null-operations. Although the capability for full MIMD operation exists, it is not studied in this research in order to focus on the external/SIMD-internal/MIMD paradigm. In contrast with the architecture described previously, here the topology of a parallel group is not constrained to a column segment because PEs do not have to be on different rows to receive different microinstructions. This makes the architecture vastly more reconfigurable and flexible. To support inter-PE communication, however, PEs in a parallel group should still be adjacent in the four NEWS directions. Additionally, the CONDITION register is vital while the DISABLE register serves no useful purpose. Also, the number of instruction cycles required to perform a branch is only equal to the number of cycles required for the longest branch. This almost certainly results in a drastic reduction in the total number of cycles required to perform a given operation.

This architectural concept is definitely the product of both miscommunication and creativity. The tradeoff for the considerable performance and flexibility advantages is that more hardware is required since the number of MMUs increases from one to the number of PEs. In fact, each PE becomes a simplistic microprocessor. If performance and reconfigurability are given priority, then this design represents a

valid compromise. It is this positive view that has been taken with this research.

Lastly, the amount of time required to adequately perform all of the experiments desired turned out to exceed that which was available, so the number of experiments actually performed is somewhat less than the original goal. Thus it is that floating-point division remains unstudied, while the study of floating-point addition/subtraction is incomplete.

To aid in identifying the various model programs and files that were developed, appear in the Appendix, and are referred to in the following discussions, Table IV is provided. It lists the file names, the modules contained, and the PE modules instantiated, and also gives brief descriptions of each.

The order in which the items are listed in the table provides a convenient organization for the following subsections which describe the configuration models and present their results. It also reflects the organization of the Appendix, and can therefore serve as its index. Discussion of the results and various tabular and graphical representations are presented in the "Analysis" section which follows this one.

Table IV

Behavioral Model Program Files

File	Module	PE	Description
pe.v	pe0	—	first PE model, least features
	pe	—	adds multi-destination memory bus
	pe2	—	adds variable-shift to EAST & WEST regs
	pe3	—	adds inversion to carry unit
	pe4	—	adds selectable-destination to carry unit, selectable source to EAST & WEST regs, removes logic/carry mutual exclusion rule
pc01–0.v	pc1	pe0	add, int, unsigned, no carry-op inversion
pc01–v	pc1	pe	add, int, unsigned, no carry-op inversion
pc02–0.v	pc2	pe0	mult, int, unsigned, 2W res, 1 PE
pc02–.v	pc2	pe	mult, int, 2s-compl, 2W res, 1 PE
pc03–.v	pc3	pe	mult, int, 2s-compl, 1W res, 1 PE
pc04–.v	pc4	pe	mult, int, 2s-compl, 1W res, 2 PEs, 1bit-shift
pc05–2.v	pc5	pe2	mult, int, 2s-compl, 1W res, 2 PEs, vari-shift
pc06–2.v	pc6	pe2	mult, int, 2s-compl, 1W res, 4 PEs, vari-shift
pc07–2.v	pc7	pe2	mult, int, 2s-compl, 1W res, 8 PEs, vari-shift
pc08–.v	pc8	pe	div, int, 2s-compl, 1 PE, restoring
pc09–.v	pc9	pe	div, int, 2s-compl, 2 PEs, restoring
pc10–.v	pc10	pe	div, int, 2s-compl, 1 PE, non-restoring
pc11–.v	pc11	pe	div, int, 2s-compl, 2 PEs, non-restoring
pc12–3.v	pc12	pe3	add, int, 2s-compl, carry-op inversion
pc13–3.v	pc13	pe3	add, fp, 32-bit, 1 PE
pc14–3.v	pc14	pe3	add, fp, 64-bit, 1 PE
pc15–3.v	pc15	pe3	mult, fp, 32-bit, 1 PE
pc16–3.v	pc16	pe3	mult, fp, 64-bit, 1 PE
pc17–3.v	pc17	pe3	mult, fp, 32-bit, 2 PEs
pc18–3.v	pc18	pe3	mult, fp, 64-bit, 2 PEs
pc19–3.v	pc19	pe3	mult, fp, 32-bit, 4 PEs
pc20–3.v	pc20	pe3	mult, fp, 64-bit, 4 PEs
pc21–3.v	pc21	pe3	mult, fp, 32-bit, 8 PEs
pc22–3.v	pc22	pe3	mult, fp, 64-bit, 8 PEs
pc23–3.v	pc23	pe3	add, fp, 32-bit, 2 PEs, incomplete

Integer Addition/Subtraction

Since the full addition/subtraction capability is designed into the PE architecture as previously described, this subsection is only presented for completeness. As previously shown, a full integer addition/subtraction requires exactly 2 cycles, one for the carry-word computation and one for the Boolean addition. If loading and storing to memory is included, this number increases to 5 -- two to load each of the operands and one to store the result. This is irrespective of the PE's specified data word width, and clearly, the number of cycles cannot be reduced by the application of multiple processors.

As can be seen from Table IV, three "configuration" model programs were written for integer addition/subtraction, mostly to test the PE model module programs and provide practice for programming configuration models. They all employ different versions of the PE model, so they can be used to present a clear example of how the PE design evolved. The programs can be seen in the Appendix, and will not be discussed further since they do not contribute any significant results.

Integer Multiplication

There are a number of varieties of integer multiplication from which to choose: signed or unsigned, single least- or most-significant word or double-word result. As indicated in Table IV, research was initiated arbitrarily with the unsigned double-word result, was changed to 2s-complement signed double-word result for greater general applicability, and was changed again to 2s-complement signed single least significant word result for intended increased floating-point applicability (although it was later realized that floating-point multiplication uses the most significant word). There were three

configuration model programs written for the two double-word result cases, all containing a module "pc2". They can be seen in the Appendix, and will not be discussed further since they do not contribute significantly to the results of the experiment.

That leaves the 2s-complement signed multiplication which produces a single-least-significant-word result. The method used is the common, straightforward shift-and-add process observed when performing manual multiplication. In this case, it was decided to shift the "bottom" operand in order to examine each of its bits individually, add the "top" operand to the partial-sum if the previously examined bit was a 1, and then shift the "top" operand left to prepare for the next add. A more detailed version of the algorithm is shown below in a "pseudo-code" form organized to accommodate the particular architecture of the PE.

```
make operands positive if necessary;
for every bit of a data word:
      rotate operand #1 right 1 bit;
      if new msb of operand #1 is 1:
            add operand #2 to lsw of partial sum;
            (end if)
      else:
            execute null-operations to synchronize;
            (end else)
      shift operand #2 left 1 bit;
      (end for)
adjust sign of result if necessary;
```

There are a number of special things to notice about this algorithm. First, overflow detection is not shown because the responsibility for it is given to the PE's MCU, but it is programmed into the module. Second, the way the bits of a data word are examined from least to most significant is by first rotating the word one bit to the right, so that the lsb becomes the msb, and then examining the msb. That turns out to be the best way to do it with this PE design. Third, all the possible bits in the data word are processed, and the "if"

statement is followed by an "else" statement that takes up the same number of cycles. This is to satisfy the external-SIMD paradigm discussed previously where all machine operations must take the same number of cycles. Fourth, the signs of the operands and result are explicitly adjusted. Referring to Table IV, the reason for this can be seen to be that the ability to invert the operands of a carry-word computation did not yet exist in the PE design. As previously described, that feature permits the negation of a data word in exactly two cycles. Examination of the programs reveals that the microinstruction cycles used for 2s-complementation where the carry-operand inversion feature is lacking can be precisely identified. Therefore, the difference in cycles can be accurately determined without reprogramming those configurations, and the effect of the additional carry unit complexity can be readily quantified. Finally, notice that the data word width is a parameter, so performing experiments for different widths is particularly easy.

That description fits the single-PE integer multiplication model, the module "pc3" in file "pc03-.v" shown in Table IV. The simulation requires 142 cycles for the 32-bit multiplication and 270 for 64 bits. This includes loading from and storing to memory, and can be broken down into a 4-cycle loop and 14 other cycles. As previously mentioned, the number of cycles required to adjust the signs can be determined, and it is found to be 10 of the 14. Replacing those sequences with the two-cycle negation offered by the carry-operand inversion feature and combining cycles where possible results in a total saving of 6 cycles. Thus the feature is of little benefit in this case.

The first attempt at applying parallelism can be seen from Table IV to be module "pc4" in file "pc04-.v". The concept employed involves taking advantage of the independent nature of the individual shift-and-add iterations described previously to distribute them equally among two PEs. Since word-communication is desired, the PEs are a North-South-connected parallel group. The fact that they have individual MCUs is modeled in the Verilog behavioral language by using separate "if-else" statements to set up each PE's microinstruction individually prior to clocking. The iterations are distributed by having one PE begin operation on bit position 0, having the other begin on bit 1, and having both skip one bit at the end of each iteration. In other words, one PE computes the partial sum using only the even-numbered bits, while the other uses only the odd, so the number of iterations performed by each PE is half of what it was in the previous case, or WORD_WIDTH/2. This method is natural given that the PE model used can only shift a single bit position at a time. At the end, one (or both) processors collect the partial sum from the other and add the two partial sums to get the final result. An additional opportunity for concurrency is presented when adjusting the signs of the operands before the loop. Instead of each PE adjusting both operands, they each adjust a different operand, and then simply exchange them.

The simulation requires 115 cycles for the 32-bit multiplication and 211 for 64 bits. The loop contains 6 cycles, the additional 2 resulting from the skipping of the bits in each of the two operands, and this makes it obvious that linear speedup is not possible. Non-loop cycles increased to 19, the additional 5 being required to offset the two operands before entering the loop and to transfer and add the two

partial sums after the loop. This would be worse if parallel operand sign adjustment was not used. If carry-operand inversion was available, the use of the two-cycle negation would save a total of 4 cycles, once again not particularly significant.

The obvious way to apply 4 PEs, also connected North-South in a line for word-width communication, to the multiplication operation is to extend the 2-PE application, i.e., offset the starting bit position of each of the four PEs by one, and have them all skip three cycles in the loop. This again reduces the number of iterations from the previous case by half, to WORD_WIDTH/4, but it can be seen by examining the previous 2-PE program to increase the number of loop cycles to 10, the 4 additional cycles resulting from the two more shifts of each of the two operands. Also, setting up the two additional offsets requires 4 more cycles and swapping and adding the two partial-sum-sums requires 4 more cycles, raising the number of cycles outside the loop to 27. The 32-bit multiplication therefore requires a total of 107 cycles, and 64 bits requires 187 cycles. Because the sign-adjustment is identical to the 2-PE case, the use of a carry-operand inversion feature would again save only 4 relatively insignificant cycles.

Similarly, the application of 8 PEs again reduces the number of iterations by half, to WORD_WIDTH/8, increasing the number of loop cycles to 18, and increasing the number of non-loop cycles to 40, resulting in a 32-bit multiplication requiring 112 cycles and 64 bits requiring 184 cycles. Note that for this method, the 8-PE 32-bit multiplication requires 5 more cycles than the 4-PE version, actually producing a "slowdown". Similarly, it is clear from the saving of a mere 3 cycles that the application of 16 PEs would then result in a

slowdown for the 64-bit multiplication. Like the 4-PE case, the use of a carry-operand inversion feature would still save only 4 relatively insignificant cycles.

Model programs were not created for the latter two cases because the PE design was modified. From the 2-PE case, it was clear that a significant number of cycles were being consumed by the shifting, and that the application of more PEs would make the effect more pronounced. This is indeed the case illustrated in the previous discussion. However, if a means existed to shift a large quantity of bit positions in a single cycle, one of the two PEs could begin operation halfway through the data word, eliminating both the need to skip bits in the loop and the need to perform multiple PE offset setup steps, thereby requiring fewer cycles. So the variable-barrel-shift logic was added to the EAST and WEST registers of module "pe", creating module "pe2", and the 2-PE experiment was redesigned to take advantage of the new feature in exactly the manner described, forming model "pc5" in file "pc5-2.v" seen in Table IV. Of course, since all of the bits must be shifted one way or another, variable-shifting is of no use in the single-PE case.

Examination of module program "pc5" reveals that it resembles a hybrid of "pc4" and "pc3"; the code before and after the loop is identical to "pc4" except for the new shift quantity, and the loop itself is identical to "pc3" where no bit-skipping occurs, except that the number of iterations is halved. The simulation requires 83 cycles for the 32-bit multiplication and 147 for 64 bits. The number of loop cycles is 4 and the number of other cycles is 19, both as expected. The saving of 4 cycles by using a carry-operand inversion feature appears more significant in this case.

Similarly, module "pc6" in file "pc06-2.v" resembles a combination of the non-loop code described for the previous 4-PE case and the loop from "pc3". The simulation requires 55 cycles for the 32-bit multiplication and 87 for 64 bits. The number of loop cycles is still 4 and the number of other cycles is 23, both as expected (23 is the 27 from the previous 4-PE case minus the 4 for operand setup shifts). Again, the saving of 4 cycles by using a carry-operand inversion feature appears more significant.

Also similarly, module "pc7" in file "pc07-2.v" resembles a combination of the non-loop code described for the previous 8-PE case and the loop from "pc3". The simulation requires 44 cycles for the 32-bit multiplication and 60 for 64 bits. The number of loop cycles is still 4 and the number of other cycles is 28, both as expected (28 is the 40 from the previous 8-PE case minus the 12 for operand setup shifts). Again, the saving of 4 cycles by using a carry-operand inversion feature appears more significant.

This concludes the development of integer multiplication. The results are discussed further and presented in various tabular and graphical forms in the "Analysis" section which follows this one.

Integer Division

The first attempt at dividing 2s-complement signed integers is shown in Table IV to be module "pc8" in file "pc08-.v". It uses a slightly modified version of the shift-subtract-restore method commonly described in the literature [24][25][26] where the need to restore is eliminated by delaying the store. A pseudo-code version of the algorithm is shown below, organized to accommodate the particular architecture of the PE.

```
determine if divisor is zero;
make dividend positive and divisor negative if necessary;
clear remainder;
for every bit of a data word:
      shift remainder left 1 bit with msb of dividend in;
      compute sum of remainder and negated divisor;
      if result is negative (msb of result is set):
            execute null-operation to synchronize;
            shift dividend/quotient left 1 bit with 0 in;
            (end if)
      else:
            replace remainder with result;
            shift dividend/quotient left 1 bit with 1 in;
            (end else)
      (end for)
adjust signs of results if necessary;
```

First note from Table IV that the programs are written using PE module "pe", and many of the characteristics of the integer multiplication algorithm discussed previously exist here: processing all the bits in the data word, null-operations for synchronization, explicit sign adjustment, predictable effect of carry-operand inversion, and easy data word width modification. Second, the effect of dividing by zero is specified to be unpredictable; the condition is flagged by the MCU, but the operation is allowed to proceed as usual to avoid unnecessary programming complication. Third, the divisor is negated before the loop because it would otherwise have to be performed inside the loop if a full subtraction was used instead of a full addition (recall that PE module "pe" has cannot invert a carry-operand). Fourth, the result of the subtraction is not immediately placed back into the remainder register like the traditional "restoring" division algorithm calls for. If it was, the above algorithm would be different by the presence of an addition (i.e., "restoration") of the divisor to the remainder in the "else" statement, and the absence of the remainder replacement (i.e., "store") operation from the "if" statement. This would cause the longest iteration to require two distinct full addition

operations instead of just one, clearly a more costly proposition. Fifth, the register containing the dividend is "recycled", and after all the iterations are complete, it contains the quotient. Lastly, note that the actual program performs the same addition at two points in the loop. This is because, since there is only one distinct addition and the registers containing the operands and the carry-word can remain unchanged between the first and second times that the sum is required, re-computing the sum without re-computing the carry-word requires only one cycle compared to the two otherwise required to first store and then retrieve the sum.

The simulation of model "pc8" requires 213 cycles for the 32-bit multiplication and 405 for 64 bits. As with integer multiplication, this includes loading from and storing to memory. It can be broken down into a 6-cycle loop and 21 other cycles. As was discussed previously, using the two-cycle negation offered by the carry-operand inversion feature instead of the sign-adjustment sequences can be found to produce a total saving of 8 cycles. Thus the feature is of little benefit in this case.

Because the iterations of a division operation are not independent, no straightforward way exists to apply parallelism. In this case, however, the PE architecture is the source of one serialization that is algorithmically unnecessary. Since the algorithm calls for two different variables to be shifted left, and the PE has only one register with left-shifting ability (EAST), it was hypothesized that a second EAST register provided by an additional PE might be successfully employed to reduce the number of cycles in a loop iteration. Also, the same kind of sharing of sign-adjustment duties

performed for multi-PE integer multiplication should save an additional, albeit relatively small, number of cycles.

This is the approach taken with the 2-PE model, module "pc9" in file "pc09-.v". Since the algorithm calls for one of the left-shifting registers to shift-in the bit shifted out of the other left-shifting register, the natural connection for the two PEs is East-West, so this is the configuration used. Examination of the loop used in the single-PE model reveals that 5 of its 6 cycles are serially dependent, so only 1 cycle is saved. Parallelizing the sign-adjustment saves 9 cycles leaving 12, so the simulation of model "pc9" requires 172 cycles for a 32-bit division and 332 for 64 bits. If carry-operand inversion was available, the use of the two-cycle negation would save a total of 4 cycles, not particularly significant.

No means of applying more processors to shift-subtract-restore division was found, so another algorithm was tried. This one is also commonly described in the literature, and is called non-restoring-shift-add-or-subtract [24][25][26]. Note that even though the previous algorithm was modified to be non-restoring it is still referred to as restoring, and it differs from this one which is referred to as non-restoring even though it actually performs a restore at the end. A pseudo-code version of the algorithm is shown below, organized to accommodate the particular architecture of the PE.

```
make dividend positive if necessary;
make positive and negative versions of divisor;
clear remainder;
for every bit of a data word:
      if remainder is negative (msb is set):
            shift remainder left 1 bit with msb of dividend in;
            add positive divisor to shifted remainder;
            (end if)
      else:
            shift remainder left 1 bit with msb of dividend in;
            add negative divisor to shifted remainder;
```

```
            (end else)
      if modified remainder is negative (msb is set):
            shift dividend left 1 bit with 0 in;
            (end if)
      else:
            shift dividend left 1 bit with 1 in;
            (end else)
      (end for)
if resulting remainder is negative (msb is set):
      add unnegated divisor to remainder;
      (end if)
else:
      execute null-operation to synchronize;
      (end else)
adjust signs of results if necessary;
```

The division experiment was repeated with this algorithm, producing a single-PE model in module "pc10" of file "pc10-.v" and a dual-PE model in module "pc11" of file "pc11-.v" as seen in Table IV. The results obtained with this algorithm are worse than with the previous one, so all of the details will not be discussed. However, some useful observations were made. The optimal loops designed for these models are exactly the same length as those designed for the previous algorithm, the non-loop operations only require 4 more cycles in the 1-PE case and 3 more cycles in the 2-PE case, and the only visible method of applying parallelism is the same as for the previous algorithm, so it appears that the performance is virtually identical. However, a method is employed whereby both the positive and negative versions of the divisor are stored in PE memory for access during the loop. Consideration after the fact resulted in the conclusion that storing PE operation variables in memory is undesirable because the allocation and addressing of such memory is impractical. Examination of the loops reveals that retrieving the divisors if they are stored in registers requires one additional cycle, and will certainly produce a drastic increase in the total number of cycles required to perform the division. The subsequent addition of the carry-operand inversion

feature to the PE design might impact the implementation of this algorithm, but this remains to be explored. Thus the register-storage versions of the modules were not developed; the memory-storage versions can be seen in the Appendix.

This concludes the development of integer division. No additional algorithm was encountered in the literature that appeared likely to offer a lower worst-case cycle count; this is discussed more in the "Analysis" section which follows this one, along with various tabular and graphical representations of the results.

Floating-Point Representation

The floating-point representation used here is a minor variation of the standard defined by the IEEE [25][24][26]. It can be seen from Figure 13 to consist of three fields: sign, exponent, and mantissa.

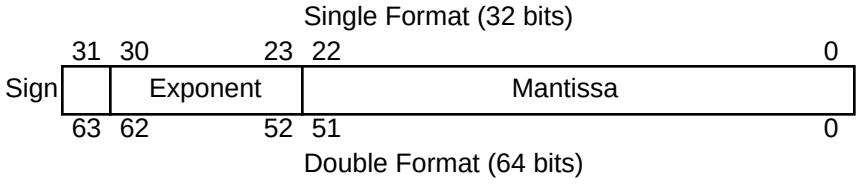


Figure 13. IEEE Standard Floating-Point Representations

The sign bit indicates the sign of the mantissa and occupies the msb of the floating-point word. The next adjacent field is a $2^{n-1}-1$ biased 2s-complement signed binary representation of the exponent, where n is the number of bits in the field. Thus for the 32-bit "single" format, the exponent field consists of 8 bits, so the bias is 127 (decimal). That means that the smallest possible exponent is -127 and

is represented by 0, a zero exponent is represented by 127, and the largest possible exponent is 128 and is represented by 255. Similarly for the 64-bit "double" format, n is 11, yielding a bias of 1023, so the smallest exponent is -1023 and is represented by 0, a zero exponent is represented by 1023, and the largest exponent is 1024 and is represented by 2047. The last field is the fractional portion of the normalized magnitude of the mantissa. A normalized mantissa is one that is expressed as a single non-zero non-fractional digit and a number of fractional digits, a condition which can be created for most values by adjusting the exponent. Note that for binary numbers, a non-zero digit must be a 1, so the non-fractional digit is thereby known and assumed instead of represented.

To maintain emphasis on the performance of the actual floating-point operation, some of the special cases defined by the IEEE floating-point standard are modified or eliminated. Additional bits (guard bit, round bit, sticky bit) are not modeled. Single- and double-extended formats are not modeled, only plain single and double formats. Special values NaN (not-a-number), ∞ (infinity), denormalized values (where the non-fractional digit is zero because the exponent cannot be made smaller), and negative zero (zero with the sign bit set) are not implemented. The special values used instead are positive and negative maximum (all bits of exponent and mantissa set), positive zero (all bits clear), and negative minimum (only sign bit set). Finally, rounding methods other than rounding-down by truncation are not used. This is the simplest and fastest method, although also the least accurate.

All the floating-point modules employ PE model "pe3" which introduces the carry-operand inversion feature. The addition of this

feature was not driven by a particular need; it was simply a general enhancement. They also benefit from a new approach to code documentation. One fact learned from the integer division experiments was that the quality and quantity of comments used in those configuration model programs was insufficient to facilitate readability and comprehension beyond the very near term. This was much less of a problem with integer multiplication than with division due to the relative difference in algorithmic complexity. The new approach uses multiple, hierarchical components and a redesigned pseudo-assembly documentation language to provide complete and organized coverage of all information relevant to the programming of the behavior. The significant difference can be observed by comparing module "pc13" to "pc11" in the Appendix. Further examination of the module programs shows that determining the cycle count that would result from the removal of the carry-operand inversion from the programs is much more difficult than doing the opposite as was done previously. Since the feature is employed in the floating-point models, the comparison is not made for floating-point. Lastly, note that all the configuration models require separate 32- and 64-bit versions. This is because some of the steps performed are so dependent on the lengths and positions of the floating-point fields, and the flexibility of Verilog's parameterization is so limited, that developing separate versions is significantly easier than developing a single universal program.

Floating-Point Addition/Subtraction

The first attempt at modeling a floating-point function can be seen in Table IV to be modules "pc13" and "pc14" in files "pc13-3.v" and "pc14-3.v". They model the single-PE addition/subtraction of 32- and

64-bit floating-point numbers, respectively. The operation is described by six stages: load and decode the operands, align the operands, add or subtract the operands, 2s-complement the result, normalize the result, and encode and store the result. The stages are discussed individually and in order below, and more detailed pseudo-code representations are included where appropriate.

The first stage is to load and decode the operands and handle the special value zero. This is straightforward, and does not merit a pseudo-code representation; the details can be seen in the Appendix. Note that the value identified as zero is handled as a special case. This is because the masking of the mantissa includes ORing in the assumed non-fractional digit of 1, but the assumption is false for the zero value (and only that value). Also, the masking operations require constants, and these are stored in memory; this is further discussed in the "Analysis" section. Note that the bias of the exponents is not removed. This is partly because the biased values, which are normally positive, are easier to work with, and partly because the msb, by indicating a negative biased exponent, indicates that the number is too small to represent in the space available in the floating-point word. Lastly, the variable-shift feature added with PE model "pe2" is almost a necessity for decoding the exponent. Without it, the exponent would either have to be right-justified by performing a large number of 1-bit shifts, or more likely, used unjustified, increasing the difficulty of programming.

The second stage is to align the operands. Alignment refers to "moving" the decimal point of the smaller operand to the same absolute position as the larger operand by adjusting the exponent. Such

alignment is clearly necessary in order not to lose precision, that being the entire point of all the floating, i.e., the floating-point. The algorithm follows.

```
determine which exponent is smaller, and set up;
for every bit of a mantissa:
      if the "smaller" exponent is really smaller:
            increment the smaller exponent;
            shift its mantissa right 1 bit;
            (end if)
      else
            execute null-operation to synchronize;
            (end else)
      (end for)
use the larger exponent for the result exponent;
```

To make the loop as small as possible, shortcuts are used. The determination of which exponent is smaller and the placement of the exponents and mantissa into the working registers is performed before the loop. Then inside the loop, since it is known which register contains the smaller exponent, it only remains to be determined if and when the two exponents become equal, and this can be done with a simple exclusive-or instead of an expensive full subtraction.

The third stage is to add or subtract the operands. This is not as straightforward as it appears. The actions are better described by a truth-table than by an algorithm, so this is presented below as Table V.

Table V

FP Addition/Subtraction Operations and Signs

Operand #1 Sign	Operand #2 Sign	Specified Operation	Perform Operation	Result Sign
+	+	+	Op1 + Op2	+
+	+	–	Op1 – Op2	?
+	–	+	Op1 – Op2	?
+	–	–	Op1 + Op2	+
–	+	+	Op2 – Op1	?
–	+	–	Op1 + Op2	–
–	–	+	Op1 + Op2	–
–	–	–	Op2 – Op1	?

From the table it can be seen that there are four distinct combinations of required operations and result signs, so these must be executed based on the three variables. In the Verilog behavioral program, this is done using a 4-stage if-else. The question-marks indicate that the required result sign cannot be determined empirically and must therefore be taken from the actual result of the operation.

The fourth stage is to 2s-complement the result mantissa if the addition/subtraction made it negative, and is required since the floating-point representation specifies an unsigned mantissa. This is straightforward, and does not merit a pseudo-code representation; the details can be seen in the Appendix.

The fifth stage is to get the result into the normalized form described previously. The algorithm for doing this follows.

```
left-justify result mantissa bitspace by shifting left;
increment result exponent;
for every bit of a mantissa:
      if result mantissa is not normalized:
            if result mantissa msb is 0:
                  decrement result exponent;
                  (end if)
            else
                  execute null-operation to synchronize;
                  (end else)
            shift result mantissa left 1 bit;
            (end if)
      else
            execute null-operation to synchronize;
            (end else)
      (end for)
```

The goal here is to find the most-significant digit of the mantissa, and then make it the non-fractional digit and adjust the exponent accordingly. The easiest way to do that with this PE architecture is to examine the msb, so the first task is to left-justify the space that the mantissa occupies. Note that the addition of two binary values of a given size may produce a result that is larger by one

msb, so the left-justification must account for the additional mantissa space. If the mantissa value actually fills the added msb position, then the result exponent, being the larger of the two operand exponents, is therefore one bit too small and must then be incremented. Similarly, if the mantissa is smaller, the result exponent is too large and must then be decremented. The best way to keep the loop simple is to make it uniform by only using it for decrementing, and to increment the exponent once before the loop and then start the loop at the additional mantissa msb. This is done here.

The sixth and last stage is to encode the floating-point word and handle the special cases of zero, overflow, and underflow. This is straightforward, and does not merit a pseudo-code representation; the details can be seen in the Appendix. The floating-point word is given the value of zero when the previous normalization produces a zero non-fractional digit for the mantissa and a negative exponent. It is given the maximum magnitude when the result is determined not to be zero and the result exponent exceeds its allotted number of bit positions. Otherwise, the exponent and mantissa are shifted and masked into the floating-point word. Finally, if the result is determined not to be zero, the appropriate sign bit is shifted in. This concludes the description of the floating-point addition/subtraction models "pc13" and "pc14".

The simulations require 204 cycles for the 32-bit data word and 407 cycles for 64 bits. The load and decode stages require 10 cycles for both models. Alignments require 100 cycles for the 32-bit word (24-bit operand mantissa, including the non-fractional digit) and 216 cycles for the 64-bit word (53-bit operand mantissa); the loop contains 4

cycles, and the non-loop cycles also number 4. The addition/subtraction and complementation stages require a total of 8 cycles in both models. Normalizations require 78 cycles for the 32-bit word (25-bit result mantissa) and 165 cycles for the 64-bit word (54-bit result mantissa); the loop contains 3 cycles, and the non-loop cycles also number 3. The encoding stage requires 8 cycles for both models.

No way to significantly parallelize floating-point addition was initially seen. A minor parallelization of alignment could be accomplished by using a second PE to perform the exclusive-or in parallel with the carry-word computation, saving 1 of the 4 cycles in the alignment loop, but this was deemed to not be worth the effort at this point. The problem appeared to be that the cycle-laden loops of the alignment and normalization processes were inherently sequential. However, after the completion of the floating-point multiplication experiments, the concept of alignment and normalization as searches was discovered, and this does permit parallelization. In particular, it was noticed that alignment and normalization are searches for the particular conditions that would allow the processes to stop. Unlike division, while each iteration depends on the fact that the previous iteration occurred, alignment and normalization do not depend on the results produced by previous iterations because those results are well-known. As such, parallel search techniques are applicable. The problems can be divided equally (or as equally as possible) among the PEs, the PEs can be individually set up as if the part of the search performed up to their individual starting points had resulted in failure, and the results of the first processor to find success can be used. The beginnings of the parallelization of module "pc13" exist in module

"pc23" in file "pc23-3.v", started after the completion of the floating-point multiplication experiment. Unfortunately, time did not permit the completion of "pc23", much less of the experiment, which requires at least six programs.

This concludes the development of floating-point addition/subtraction. No studies of worst-case alignment or normalization speedup techniques that might be used to provide additional experiments was known to exist in the literature. The figures presented above are tabulated in the "Analysis" section which follows this one.

Floating-Point Multiplication

As can be seen from Table IV, eight models cover the spectrum of floating-point multiplication experiments across data-word size and number of PEs: modules "pc15" through "pc22 in files "pc15-3.v" through "pc22-3.v". The multiplication algorithm is described by three stages: load and decode the operands and add the exponents; multiply the mantissas; and normalize, encode, and store the result. The stages are discussed individually and in order below, and more detailed pseudo-code representations are included where appropriate.

The first stage is to load and decode the operands, determine the result sign, and handle the special value zero. This is straightforward, and does not merit a pseudo-code representation; the details can be seen in the Appendix. Note that since decoding the mantissa is the same as for division, zero must be handled as a special case for the same reason -- the OR-masking prepends an undesired 1 non-fractional digit to it.

The second stage is to multiply the operands. This uses the same shift-add procedure described for integer multiplication, with one major difference. The goal of floating-point is to maintain the maximum precision possible, and this is accomplished by retaining the largest number of msbs that can fit into a mantissa space. Unfortunately, the form of product chosen previously for integer multiplication retains only the lsbs, making application of the previously-designed microinstruction sequences to the floating-point programs practically impossible. So the following algorithm was devised.

```
for every bit in a mantissa:
      rotate mantissa #1 right 1;
      if msb of rotated mantissa is 1:
            add mantissa #2 to result mantissa;
            (end if)
      else
            execute null-operation to synchronize;
            (end else)
      shift result mantissa right 1 bit;
      (end for)
```

Note that because both operand mantissas have 1s in the msb position (non-fractional digit), the size of the result mantissa must either be the maximum possible, which is one bit larger than the size of a mantissa, or one bit smaller, which is the same size as a mantissa. This is makes retaining the bits shifted out of the result mantissa unnecessary. Also, even though a full addition is performed, the carry-in of the computed carry-word is always zero because the largest operand is significantly smaller than a register, which is at most one bit larger than the size of a mantissa.

The third and last stage is to normalize, encode, and store the result. This is straightforward, and does not merit a pseudo-code representation; the details can be seen in the Appendix. Note that because only two sizes are possible for the result mantissa as discussed

previously, normalization is trivial. Special cases are also handled here, just as with floating-point division: zero, overflow, and underflow. The floating-point word is given the value of zero if either of the two operands were previously determined to be zero or the result exponent is negative. It is given the maximum magnitude when the result is determined not to be zero and the result exponent exceeds its allotted number of bit positions. Otherwise, the exponent and mantissa are shifted and masked into the floating-point word. Finally, if the result is determined not to be zero, the appropriate sign bit is shifted in. This concludes the description of the single-PE floating-point multiplication models, "pc15" and "pc16".

Because the loop represents the same shift-and-add method used for integer multiplication, parallelism is applied in exactly the same manner, by distributing the independent loop iterations as equally as possible among a number of North-South-connected PEs. The discussion of the parallelization of integer multiplication applies so well to the floating-point experiments that repeating it entirely here is unnecessary; the details are available from the module programs in the Appendix. There are only a few minor differences that bear mention. First, only the method relying on the variable-shift capability is used. Second, the floating-point load-and-decode stage is parallelized in the same manner as the integer sign-adjust stage, by distributing the operands among the PEs. Third, because the size of the mantissa for the 64-bit format is not evenly divisible by 2, a small amount of complexity was added to the multiple-PE 64-bit model programs. Lastly, operational clarity is enhanced by placing the reduction of the different PE partial sums in a separate stage before the normalize, encode, and store stage

instead of combining them. This concludes the description of the multiple-PE floating-point multiplication models, "pc17" through "pc22".

The simulation of the 32-bit 1-PE model "pc15" requires 144 cycles: 24 iterations of a 5-cycle loop and 24 other cycles. The 64-bit version "pc16" requires 289 cycles, the only difference being that the number of iterations increases to 53. The 32-bit 2-PE model "pc17" requires 90 cycles: 12 iterations of the 5-cycle loop and 30 other cycles. The increased overhead consists of 1 more cycle to set up the shifted operand for the loop, and 5 cycles to reduce the PE partial sums. The 64-bit version "pc18" requires 165 cycles; 26 iterations of the loop cover 52 bits, while the 53rd is processed by a "special iteration" performed by 5 post-loop microinstructions. The 32-bit 4-PE model "pc19" requires 64 cycles: 6 iterations of the 5-cycle loop and 34 other cycles. The increased overhead consists of 4 more cycles for the PE partial sum reduction. The 64-bit version "pc20" requires 104 cycles and 13 loop iterations. The 32-bit 8-PE model "pc21" requires 54 cycles: 3 iterations of the 5-cycle loop and 39 other cycles. Again, the increased overhead consists of 4 more cycles for the PE partial sum reduction. The 64-bit version "pc22" requires 74 cycles: 6 iterations of the loop cover 48 bits, and the remaining 5 are processed by a "special iteration" that is 1 cycle longer than that required in the 4-PE case.

This concludes the development of floating-point multiplication, as well as this section, "Arithmetic Performance". The results that were mentioned are discussed and presented in various tabular and graphical forms in the following "Analysis" section.

ANALYSIS

Overview

As previously indicated, the foremost goal if this research is to characterize the suitability of this PE and system architecture for speeding up arithmetic operations through the application of parallelism. Note that arithmetic machine operations are, by nature, fixed-size problems. Therefore, the two most informative forms of the "speedup" of a given operation for this study are

$$S[P] = T[1] \div T[P]$$

and

$$S'[P] = S[P] \div P \quad ,$$

where P is the number of PEs applied to the operation, T is the amount of time (i.e., number of cycles) required to perform the operation given the number of PEs applied to it, S is the total or absolute speedup achieved by that particular number of PEs, and S' is the relative portion of that speedup "contributed" by each PE in the group. The ideal situation is where every PE applied to a problem justifies its cost by providing a directly proportional increase in performance; in other words, P PEs should ideally perform a given operation in 1/Pth the number of cycles required by 1 PE. Absolute speedup does not show this performance per price ratio, only the total performance gain, so the two forms of speedup are used to yield as much information as possible. Note however that factors such as the distribution of overhead among multiple PEs technically allows speedup greater than the ideal, thereby clouding the definition of "ideal" and making it more subjective.

This section presents and discusses the results of the arithmetic performance experiments in tabular and graphical form. The order is the same as was used previously: integer multiplication, integer division, floating-point addition/subtraction, and floating-point multiplication.

Integer Multiplication

The results of the integer multiplication experiments are listed in Table VI.

Table VI

Integer Multiplication Performance

Word Width	Number of PEs	Variable Shift	Loop Cycles	Other Cycles	Total Cycles	Absolute Speedup	Relative Speedup
32	1	N/A	4	14	142	1.00	1.00
32	2	No	6	19	115	1.23	.62
32	4 *	No	10	27	107	1.33	.33
32	8 *	No	18	40	112	1.26	.16
32	2	Yes	4	19	83	1.71	.86
32	4	Yes	4	23	55	2.58	.65
32	8	Yes	4	28	44	3.23	.40
64	1	N/A	4	14	270	1.00	1.00
64	2	No	6	19	211	1.28	.64
64	4 *	No	10	27	187	1.44	.36
64	8 *	No	18	40	184	1.47	.18
64	2	Yes	4	19	147	1.84	.92
64	4	Yes	4	23	87	3.10	.76
64	8	Yes	4	28	60	4.50	.56

* calculated

The column headings are self-explanatory. Note the cases marked with asterisks are those which were not actually modeled but were accurately calculated as described in the previous discussion. Note also that as previously mentioned, the figures for absolute speedup best quantify the overall performance gain, while those for relative speedup best indicates the ratio of performance to cost achieved.

Composite graphs of the absolute and relative speedups appear in Figures 14 and 15, respectively.

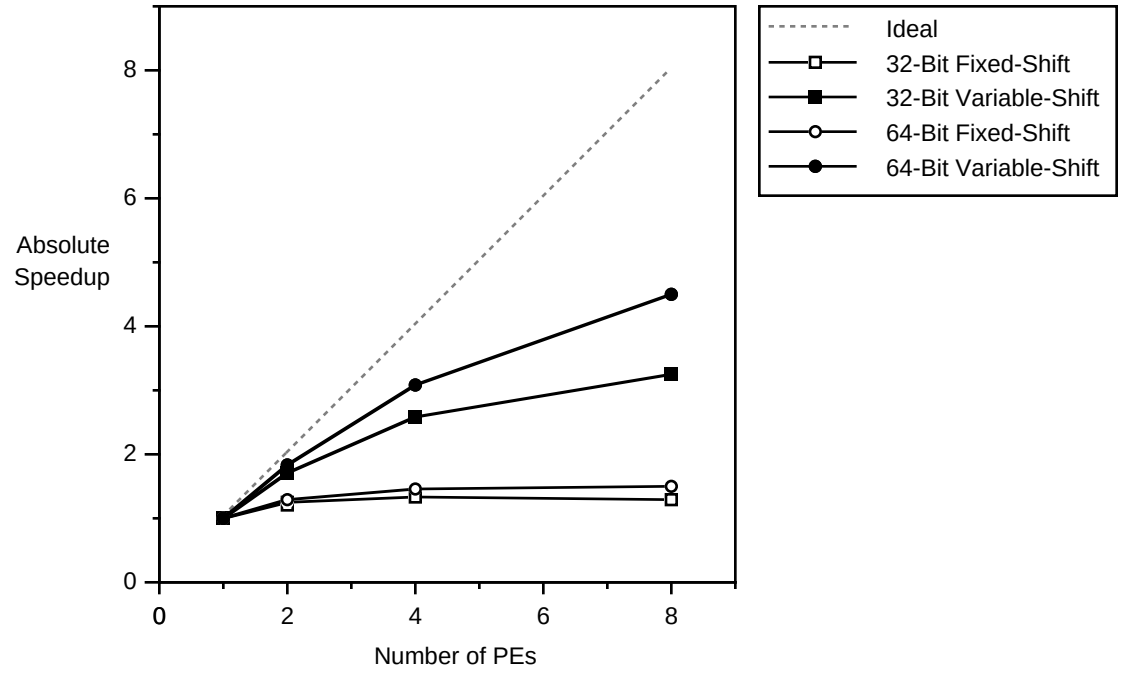


Figure 14. Absolute Speedup of Integer Multiplication

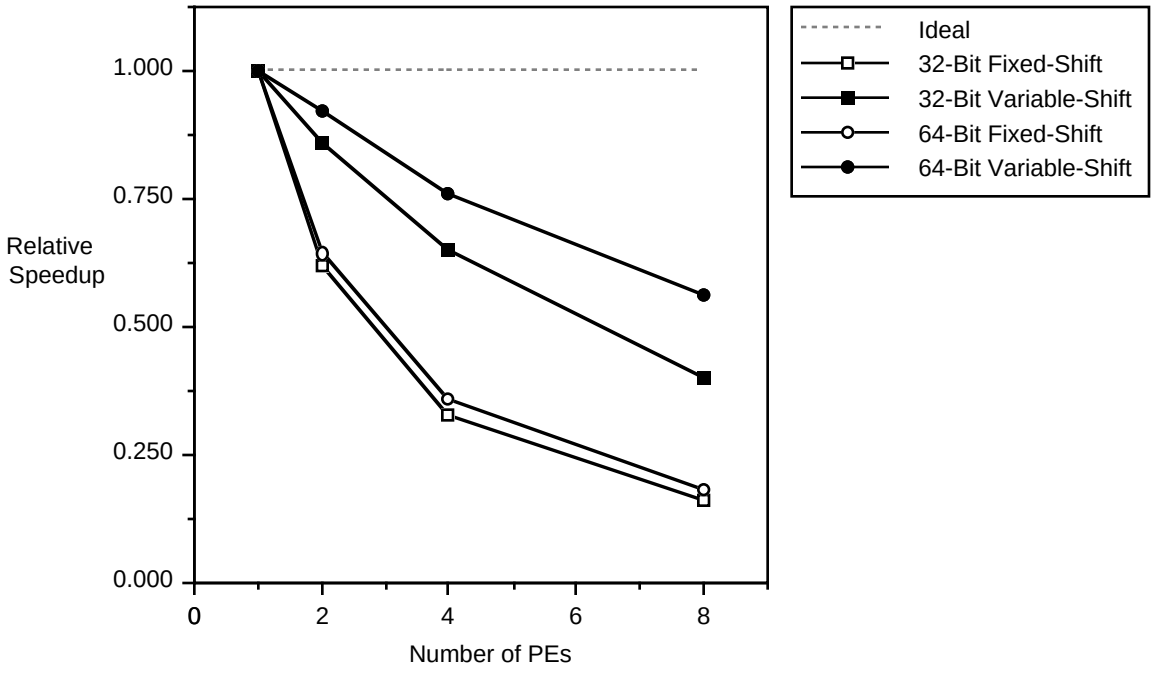


Figure 15. Relative Speedup of Integer Multiplication

A number of observations can be made from these figures. First, it is readily apparent that the ability to shift more than a single bit position in a cycle is vital to parallelizing multiplication, as was discussed previously with the performance. This is evident both from the plots and from the number of loop cycles listed in the table. Second, the relationship between the loop and the non-loop cycles is clearly visible from both the difference between 32- and 64-bit performance curves and the dropoff of the absolute speedup curves. As expected from Amdahl's Law [24], distributing the loop iterations among a larger number of PEs makes the slowly-growing number of non-loop cycles more significant. Third, the previous performance discussion established that adding a carry-operand inversion feature would save 4 cycles on most of the programs, and it can be seen from the table that the variable-shift, 32-bit 4- and 8-PE configurations and the variable-shift 64-bit 8-PE configuration would show almost a 10% improvement in performance.

It can be reasoned that these configurations are fairly close to optimal for this PE architecture. The architecture prohibits a sufficient amount of work from being done with a loop of less than 3 cycles, and these loops (with the variable-shift feature) take only 4. Due to the simplicity of the algorithm, it can be fairly stated that no change in algorithm is likely to produce significantly better performance. Logically, the application a number of PEs greater than the number of loop iterations to the operation cannot further increase performance, and the plots indicate that the maximum performance is unlikely to be significantly greater than that achieved by, perhaps, 4 PEs for the 32-bit data word and 8 PEs for 64 bits. Therefore, the only

likely source of significant additional speedup is architectural change. The carry-operand inversion feature is one such change, but one that only has significant impact when more PEs are used. Another is the removal of the logic/carry exclusion rule, i.e., using the latest PE design described previously. This may allow an algorithm to be designed with a smaller loop. If so, that would almost certainly be the performance limit for integer multiplication using this type of PE.

Integer Division

The results of the integer division experiments are listed in Table VII.

Table VII

Integer Division Performance

Word Width	Type	Number of PEs	Loop Cycles	Other Cycles	Total Cycles	Absolute Speedup	Relative Speedup
32	R	1	6	21	213	1.00	1.00
32	R	2	5	12	172	1.24	.62
32	NR	1	6	25	217		
32	NR	2	5	15	175		
64	R	1	6	21	405	1.00	1.00
64	R	2	5	12	332	1.22	.61
64	NR	1	6	25	409		
64	NR	2	5	15	335		

"Type" refers to the division algorithm used, as described previously: "R" for restoring, and "NR" for non-restoring. The speedups for the non-restoring technique are not listed because they are very similar to the restoring case, and would serve little other than to cloud the discussion. Also, as mentioned previously, the way that the algorithms were implemented is deemed undesirable, so the speedup values are not of much interest.

Composite graphs of the absolute and relative speedups appear in Figures 16 and 17, respectively.

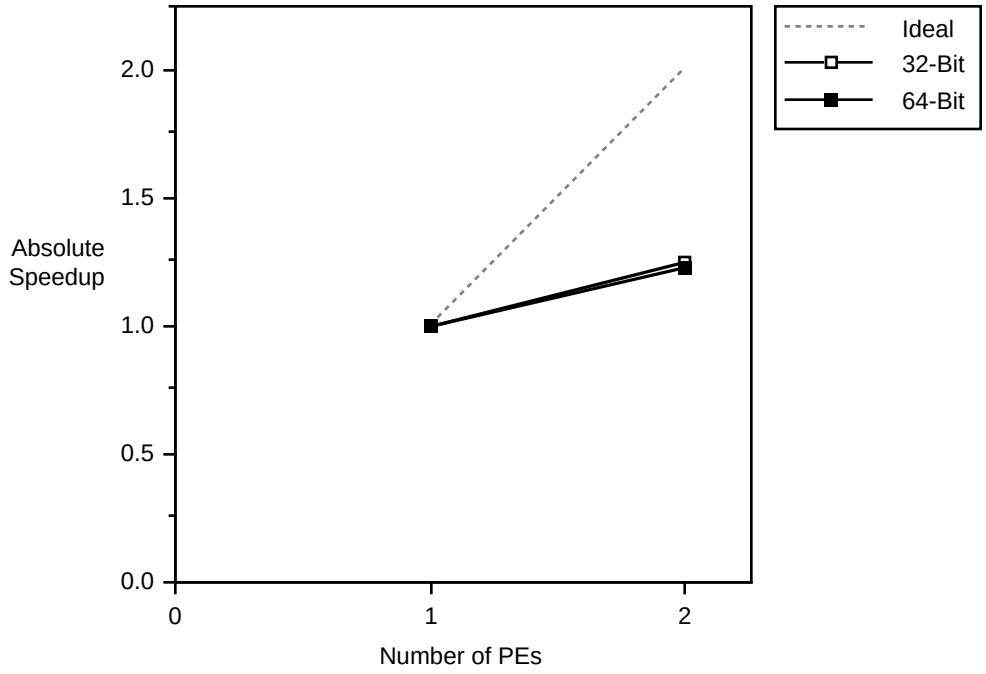


Figure 16. Absolute Speedup of Integer Division

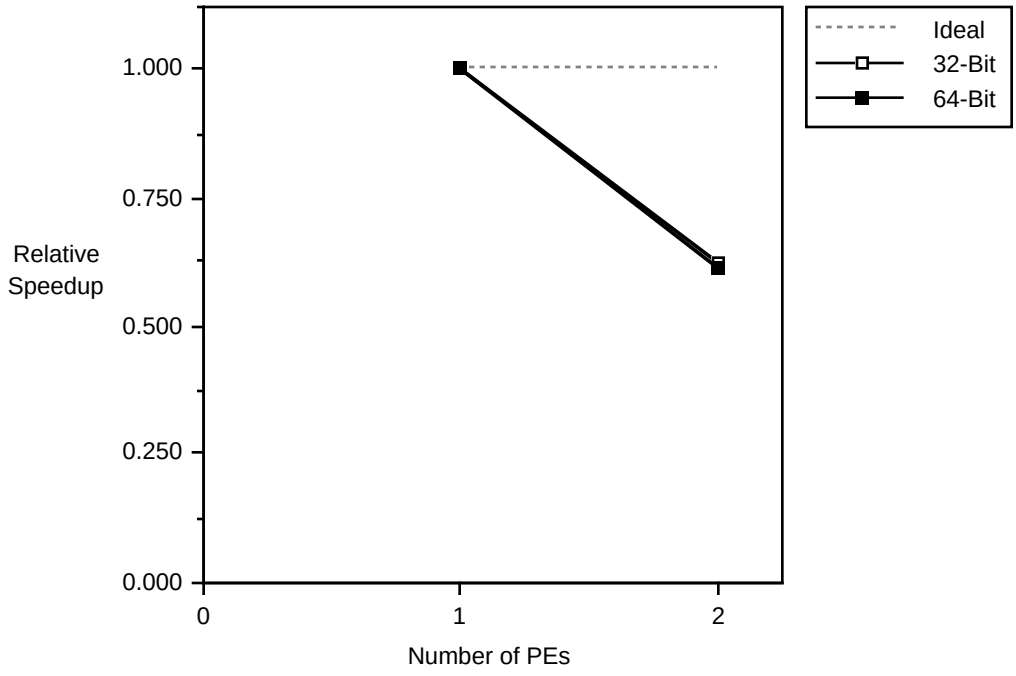


Figure 17. Relative Speedup of Integer Division

With only 2 PEs and no scalable parallelization method, the graphs are of limited use. They only show the limited effects of reducing the loop size by 17%, and are presented mostly for completeness.

It is clear that architectural adjustments could improve only the 1- and 2-PE cases but still not enable the application of scalable parallelism. For example, if the PE had a means of moving data from a selectable-source register (particularly the shiftable EAST register) to one of the main logic input registers (register IN1 in this case) in a single cycle, then the loop in the 2-PE restoring model could be reduced by one. This can be seen from module "pc9" in file "pc09-.v" in the Appendix. Also, in the same manner as multiplication, a division algorithm written for the PE design that eliminates the carry/logic exclusion rule may be more efficient. The lack of parallelizability, however, originates from the iteration dependence of the algorithm, and will not be affected by these changes.

A brief investigation was made into other division algorithms. Non-restoring division was found to be unhelpful mostly because it is actually a simple variation of the restoring algorithm, and offers no new methods. Cavanagh [25] describes a number of division methods. SRT (Sweeny, Robertson, Tocher) division works by skipping sequences of 1s or 0s, but the speedup is a result of the operand characteristics, a property not allowed in the SIMD paradigm since all operations must use the same number of cycles. The only way to achieve that with SRT is to assume worst case, and this provides no speedup. Other approaches to division uses multiplication, addition, and subtraction to converge on a result. Divisor Reciprocation, as its name implies, computes the reciprocal of the divisor using some iterative method such as Newton-

Rhapson Iteration or Goldschmidt's Algorithm [24] and then obtains the result by multiplying it with the dividend. Division Through Multiplication [27] is a method that uses a lookup table to iteratively provide fractional values to multiply with both the divisor and the dividend. It may be that these methods could provide the parallelism sought. However, these methods are all relatively complex, and a number of issues must be addressed before their usefulness could be ascertained. For example, it is unclear with these methods what the worst-case cycle count is, or even how to go about making such a determination, yet this information is vitally important to SIMD design. Also, they all contain some use of fractional binary numbers, and it remains to be determined what manner of overhead is required to track the decimal points, as well as what the cost of any additional required precision might be. It is apparent that integer division is an entire research area unto itself, and so it was elected to proceed to the study of floating-point arithmetic.

Floating-Point Addition/Subtraction

The results of the floating-point addition/subtraction experiments are listed in Table VIII.

Table VIII

FP Addition/Subtraction Performance

Word Width	Number of PEs	Alignment Loop Cycles—Iterations		Normalization Loop Cycles—Iterations		Other Cycles	Total Cycles
32	1	4	24	3	25	33	204
64	1	4	53	3	54	33	407

Since the experiment is incomplete, no particularly significant analysis is possible. One observation is that floating-point addition/subtraction requires almost exactly the same number of cycles as integer division, a remarkable coincidence. Another observation, this one not related to performance but to architecture, is that in order to perform floating-point operations, constants are required for masking the floating-point word to decode and encode its constants. In these models, the required constants are stored in PE memory, but there are also other options available to consider if necessary. For example, another bit can be added to the instruction word to increase the number of registers selectable as data sources, and some of them can be made to contain the necessary constants and be read-only. Alternately, specialized hardware could be added to perform the encoding and decoding function, although this would require a more significant modification of the existing architecture in order to interface with it. Lastly, it can be reasoned that the application of parallel search techniques to the alignment and normalization stages as previously discussed should result in a high level of scalability, extremely similar to that shown by the performance curves of integer and floating-point multiplication. They should only be slightly worse in magnitude due to the dual loops and the greater amount of overhead.

Floating-Point Multiplication

The results of the floating-point multiplication experiments are listed here in Table IX.

Table IX

FP Multiplication Performance

Word Width	Number of PEs	Loop Cycles	Loop Iterations	Other Cycles	Total Cycles	Absolute Speedup	Relative Speedup
32	1	5	24	24	144	1.00	1.00
32	2	5	12	30	90	1.60	.80
32	4	5	6	34	64	2.25	.56
32	8	5	3	39	54	2.66	.33
64	1	5	53	24	289	1.00	1.00
64	2	5	26	35	165	1.78	.88
64	4	5	13	39	104	2.78	.69
64	8	5	6	44	74	3.91	.49

Note that these results bear a striking resemblance to those obtained for the integer multiplication (with variable-shift). This is because floating-point multiplication is really just integer multiplication with fewer loop iterations but more "overhead". Because of this, the previous analysis of integer multiplication is equally applicable to floating-point. It was unfortunate that the type of product selected for integer multiplication, the least-significant word, was not the one required for floating-point; otherwise, the development would have required noticeably less effort. Also note that use of the carry-operand inversion feature for integer multiplication would increase the validity of the comparison, and would widen the difference between integer and floating-point performance by improving the 4- and 8-PE model cycle counts significantly, as discussed previously. Finally, note that because the size of the floating-point mantissa is smaller than that of the integer word, the parallelization limit is reached with fewer PEs. In fact, since the 8-PE 32-bit multiplication (as well an imagined 16-PE 64-bit multiplication) require only 3 loop

iterations, a cycle or two might be saved by unrolling the loop and combining microinstructions, but this is not known for certain.

Composite graphs of the absolute and relative speedups appear in Figures 18 and 19, respectively.

Again, these results are extremely similar to those obtained with integer multiplication. In fact, close comparison reveals that these curves are only slightly worse, as would be expected by the modification of integer multiplication to require fewer loop iterations and more overhead. It can be maintained with the arguments presented for integer multiplication that these configurations are fairly close to optimal for this PE architecture. It is also possible that using the more current PE design described previously which removes the logic/carry exclusion rule could save a number of clock cycles, producing optimal configurations with even higher performance.

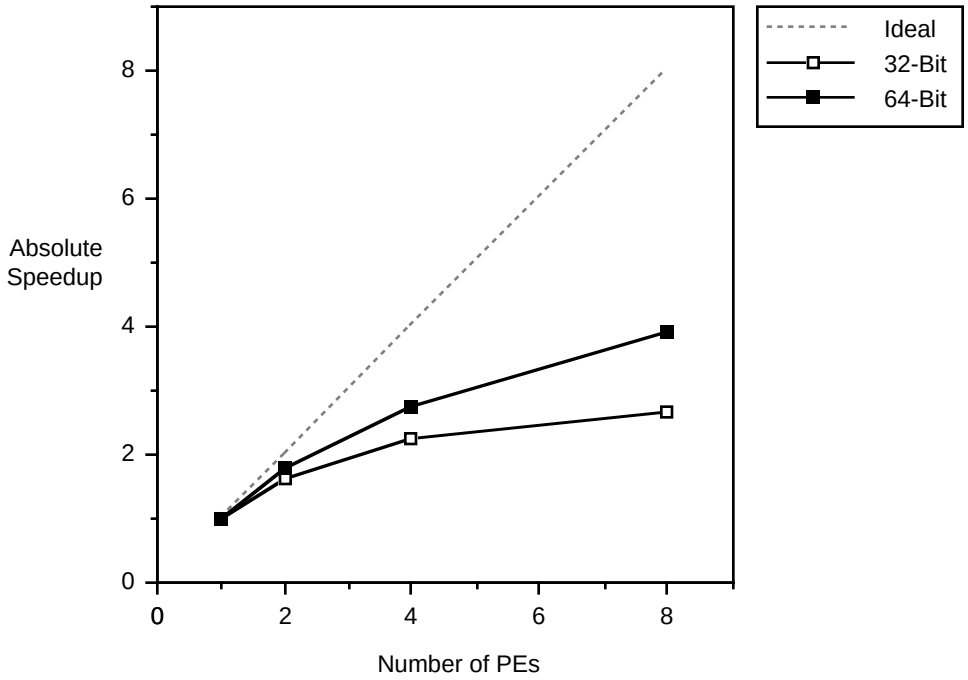


Figure 18. Absolute Speedup of FP Multiplication

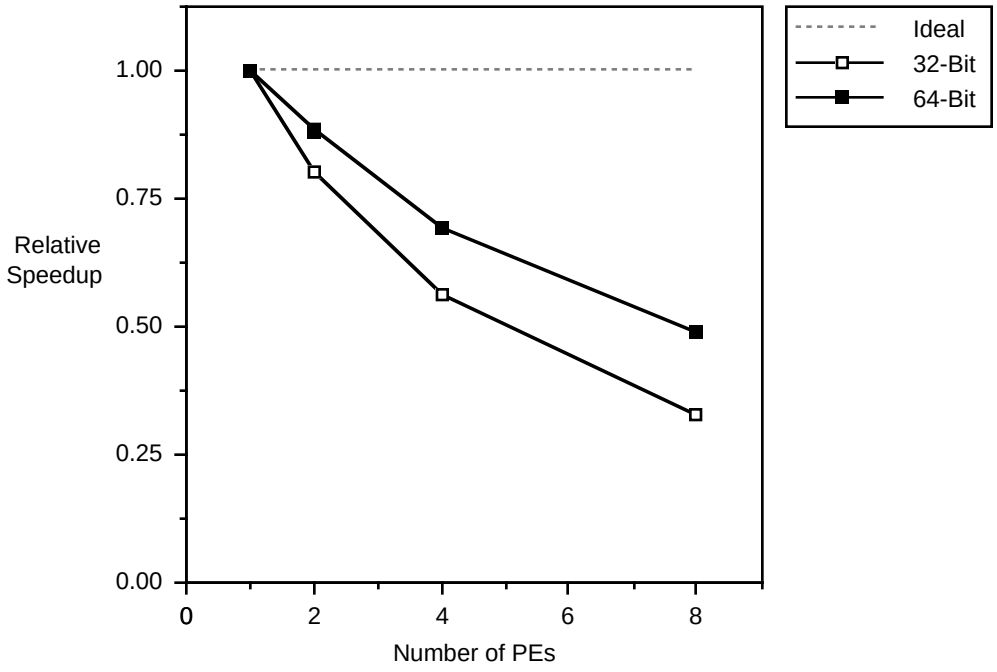


Figure 19. Relative Speedup of FP Multiplication

This concludes the analysis of floating-point multiplication, and also of the "Analysis" section. The next section concludes this work.

CONCLUSION

In this paper, a PE for a massively-parallel, rectangular-mesh-connected, method-reconfigurable, MSIMD computer, and configurations of PEs for performing various arithmetic operations, were designed and modeled. The instruction cycle of the PE can be completed in one clock cycle, so it is fast. The PE design is based on a pre-existing 1-bit-wide PE design, so it retains the advantages of being simple, small, and inexpensive. The width of the PE is the same as that of the data word, so it provides a convenient view for the application programmer. Facilities are incorporated which increase the performance of word-wide integer and floating-point arithmetic. Performance results were obtained and analyzed for configurations of 1, 2, 4, and 8 PEs, with 32- and 64-bit data word widths, applied to integer multiplication and division and floating-point addition/subtraction and multiplication. The modeling was accomplished with the behavioral component of the Verilog HDL.

The performance results showed that these arithmetic operations could be implemented efficiently with this PE design. Furthermore, they demonstrated that significant speedup could be achieved, a direct result of the parallelization afforded by the MSIMD paradigm through method-reconfigurability. Therefore, it can be concluded that a method-reconfigurable MSIMD architecture has a significant probability of providing a high-performance computer, one which is especially well-suited to applications involving large arrays of homogeneous data, at a lower cost than a comparable MIMD machine.

The most immediate continuation of this work would be to repeat the experiments without the carry/logic exclusion rule, as discussed in the Overview of the Arithmetic Performance section. This has the possibility of reducing the number of loop cycles and thereby significantly improving performance. Also, the floating-point addition/subtraction experiments should be completed, and other floating-point arithmetic operations such as division, logarithm, exponent, square-root, etc. should be investigated. Additional research into speeding up integer division could also prove beneficial.

Useful information might be provided by investigating configurations for the original MSIMD target architecture shown in Figure 11 as opposed to the alternate architecture shown in Figure 12 that was actually used. The total cycle counts would increase dramatically since the number of cycles required for an N-way branch would increase from that of the longest branch to N times that of the longest branch, but significant speedup could still be expected. For the alternate architecture, the disable feature can be removed since its function can be fulfilled by null-operations. However, some microprogramming difficulty was found to result from the fact that during a null-operation, the OUT register must be overwritten, so adding a feature to disable writing to the OUT register is worth consideration. This is not as much of a problem with the selectable-destination register because the source and destinations can be specified to be the same and the operation can be specified to be the identity, thus performing a true null-operation. Another feature that should be considered is the ability to move data from any register to the IN1/IN2 registers and/or memory in a single clock cycle. The current design

requires two cycles, one to move data from the source register to the OUT register through a Boolean logic unit, and a second to move it from the OUT register to IN1/IN2/memory. This might not only increase performance by reducing the cycle count, but could possibly free that Boolean logic unit to perform useful computation; this issue remains to be investigated.

After completion of the behavioral study, the next task would be to undertake a structural study. This could be performed using the structural component of Verilog. The microcontrollers and microinstruction memories would have to be added to the PE models, as well as at least one additional addressing mode, since direct addressing does not provide sufficient flexibility for other, non-arithmetic processor operations. After studying individual and configured PE operations, the master controller and the global data router network could be modeled, and the entire computer system could be studied. Should the results of the structural research be positive, a hardware implementation could finally be considered.

REFERENCES

1. Flynn, M. J.: "Some Computer Organizations and their Effectiveness", IEEE Transactions On Computers, vol. C-21, no. 9, September 1972.

2. Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, R. A. Stokes: "The ILLIAC IV Computer", IEEE Transactions on Computers, vol. 17, no. 8, August 1968.

3. Hockney, R. W., C. R. Jesshope: Parallel Computers, Adam Hilger Ltd., Bristol, 1981.

4. Batcher, K.: "Design of a Massively Parallel Processor", IEEE Transactions on Computers, vol. 29, no. 9, September 1980.

5. Hillis, W. D.: The Connection Machine, MIT Press, Cambridge, 1985.

6. Beetem, J., M. Denneau, D. Weingarten: The GF11 Parallel Computer, in J. J. Dongarra: Experimental Parallel Computing Architectures, North-Holland, Amsterdam, 1987.

7. Almasi, G. S., A. Gottlieb: Highly Parallel Computing, Benjamin/Cummings, Redwood City, California, 1989.

8. Kartashev, S. I. and S. P.: "Dynamic Architectures: Problems and Solutions", IEEE Computer, vol. 11, no. 7, July 1978.

9. Kartashev, S. I. and S. P.: :"A Multicomputer System with Dynamic Architecture", IEEE Transactions on Computers, vol. C-28, no. 10, October 1979.

10. Snyder, L.: "An Inquiry into the Benefits of Multigauge Parallel Computation", IEEE Proceedings of the International Conference on Parallel Processing, 1985.

11. Miller, R., V. K. Prasanna-Kumar, D. I. Reisis, Q. F. Stout: "Parallel Computations on Reconfigurable Meshes", IEEE Transactions on Computers, vol. 42, no. 6, June 1993.

12. Ligon, W. B., III: An Empirical Evaluation of Architectural Reconfigurability, doctoral thesis, College of Computing, Georgia Institute of Technology, Atlanta, August 1992.

13. Ligon, W. B., III, U. Ramachandran: "Evaluating Multigauge Architectures for Computer Vision", Journal of Parallel and Distributed Computing, vol. 21, no. 3, June 1994.

14. Snyder, L.: "Introduction to the Configurable, Highly Parallel Computer", IEEE Computer, vol. 15, no. 1, January 1982.

15. Siegel, H. J. and L. J., F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., S. D. Smith: "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", IEEE Transactions on Computers, vol. C-30, no. 12, December 1981.

16. Sejnowski, M. C., E. T. Upchurch, R. N. Kapur D. P. S. Charlu, G. J. Lipovski: "An Overview of the Texas Reconfigurable Array Computer", Proceedings of the 1980 AFIPS National Computer Conference, vol. 49, AFIPS Press, Arlington, Virginia, May 1980.

17. Choudhary, A. N., J. H. Patel, N. Ahuja: "NETRA: A Hierarchical and Partitionable Architecture for Computer Vision Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 10, October 1993.

18. Batcher, K. E.: "STARAN Parallel Processor System Hardware", Proceedings of the 1974 AFIPS National Computer Conference, vol. 43, AFIPS Press, Montvale, New Jersey, May 1974.

19. Maresca, M.: "Polymorphic Processor Arrays", IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 5, May 1993.

20. Baron, R. J., L. Higbie: Computer Architecture Case Studies, Addison-Wesley, Reading, Massachusetts, 1992.

21. Ligon, W. B., III, C. Subramanyam: "A Bit-Serial SIMD Processing Element", unpublished, College of Computing, Georgia Institute of Technology, September 1991.

22. Thinking Machines Corporation: The Connection Machine CM-5 Technical Summary, Thinking Machines Corporation, Cambridge, October 1991.

23. Cadence Design Systems: Verilog-XL Reference Manual, ver. 1.6, Cadence Design Systems, March 1991.

24. Hennessy, J. L, D. A. Patterson: Computer Architecture: A Quantitative Approach, Morgan Kaufmann, San Mateo, California, 1990.

25. Cavanagh, J. J. F.: Digital Computer Arithmetic, McGraw-Hill, New York, 1984.

26. Koren, I.: Computer Arithmetic Algorithms, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

27. Flynn, M. J.: "Very High-Speed Computing Systems", Proceedings of the IEEE, vol. 54, no. 12, December 1966.